Investigación sobre el reconocimiento de objetos físicos mediante redes neuronales

Pseudónimo: Zepto

Índice de contenidos

1. Introducción....4
2. Cuerpo. Marco Teórico....5
2.1. **Inteligencia Artificial (IA)**....5
2.2. **Aprendizaje automático (AA)**....6
2.2.1. **Aprendizaje no supervisado**....6
2.2.2. **Aprendizaje supervisado**....7
2.3. **Redes neuronales artificiales (RNA)**....11
2.4. **La esencia del backpropagation**....15
2.5. **Aprendizaje profundo (AP)**....17
2.6. **Visión por computador (VC)**....18
2.6.1. **Reconocimiento de objetos**....19
2.7. **Métodos tradicionales para la clasificación en el reconocimiento de objetos**....19
3. Parte práctica: Funcionamiento de una CNN....21
3.1. **Las redes convolucionales (CNN)**....21
3.1.1. **Imágenes**....22
3.1.2. **Primera convolución**....23
3.2. **Subsampling - maxpooling**....27
3.3. **Siguientes convoluciones**....28
3.3.1. **Transformación a red neuronal tradicional**....29
3.4. **Aplicación de la Teoría**....31
3.5. **Desarrollo de mi propuesta**....32
3.6. **El entrenamiento de la CNN**....34
3.7. **Construcción del modelo**....37
3.8. **Mis números**....39
4. Conclusión....41
4.1. **Reflexión sobre el proyecto**....42
5. Bibliografia....45
5.1. **Referencias de las figuras**....46
6. Anexos....48
6.1. **Código de la creación de la red neuronal:**....48
6.2. **Pruebas modelo CNN**....52

1. Introducción

Una de las funcionalidades que destaca a nuestro organismo, gracias a la complejidad de nuestro cerebro y sistema nervioso, es reconocer los objetos que nos rodea.

Dado el avance tecnológico en sistemas miniaturizados, si nos fijamos en los sistemas de detección que tienen los coches autónomos, nos percatamos que las máquinas son perfectamente capaces de percibir y reconocer información del entorno. Es la idea de que un ser inerte, sin ningún tipo de vínculo biológico aparente con el humano, sea capaz de obtener, interpretar y actuar en base a un objeto, la principal motivación del trabajo. Consecuentemente, ¿De qué formas una máquina puede estar casi totalmente segura de la validez de su perspectiva sobre, por ejemplo, unos caracteres, aun así, sean escritos a mano y por consiguiente tengan formas diferentes? Hago énfasis en los caracteres puesto que es un objeto con el que será mucho más fácil realizar el estudio. Entonces, ¿Qué procesos son los que sigue una máquina para poder reconocer y clasificar un carácter?

Durante el presente trabajo se pretende profundizar en el funcionamiento del reconocimiento y clasificación. En la primera parte, de carácter teórico, se busca investigar de qué maneras se ha tratado este problema, y explicar el funcionamiento de la que se relacione mejor con mi objetivo práctico. En la segunda parte, se pondrá en práctica el proyecto, basado en los contenidos explicados, con la ayuda de herramientas de programación. Finalmente, luego de haber interpretado los resultados de mi parte práctica, reflexionaré sobre la aplicación de este y las razones de por qué lo he llevado así. Paralelamente, se persiguen los siguientes objetivos:

- Haciendo un enfoque aún más concreto, se pretende ver qué método es el más adecuado para llevar a cabo una clasificación de números, del 1 al 9. Comprobar su eficiencia y ver cómo esta se podría aumentar.
- Conocer cuáles son los métodos matemáticos que se han ido proponiendo para analizar los eventos que contiene una imagen y poder dar un resultado en formato de red probabilística.

2. Cuerpo. Marco Teórico

2.1. Inteligencia Artificial (IA)

La apasionante ciencia que abarca todas estas cuestiones es la inteligencia artificial, pero, ¿qué significa realmente? Durante la Segunda Guerra Mundial, Alan Turing consiguió descifrar los códigos Nazis emitidos por una máquina, a partir de ahí, empezó a hablarse de la posibilidad que una maquina pudiera imitar la conducta humana. Esta tiene sus raíces la lógica matemática.

La informática en ese entonces solo servía para hacer operaciones matemáticas, reproducir un sonido... El gran cambio que causa la *IA* es que el propio programa *decide* cómo tratar unos datos de entrada (*inputs*) para dar unos datos de salida (*outputs*).

Al fin y al cabo, es exactamente lo que hace nuestro cerebro. Al nacer, no tenemos ningún conocimiento sobre el entorno. Necesitamos bastantes años para aprender a caminar, hablar y razonar; siempre cometiendo muchos fallos inicialmente, tendiendo así a una mejora con el tiempo. Así es como se desarrolla también una *IA*: aprendizaje, entrenamiento y resultados. Mediante algoritmos y modelos matemáticos se hacen predicciones.

Dentro de esta ciencia, el estudio que se dedica justamente a cómo llevar a cabo una tarea concreta es el **aprendizaje automático.**

2.2. Aprendizaje automático (AA)

El procedimiento que sigue un alumno (y una *IA*) es generalizar a partir de su experiencia. Construimos un modelo general sobre el conjuntos ejemplos de entrenamiento, que permita hacer predicciones suficientemente precisas.

Dentro de la informática teórica, la teoría del aprendizaje computacional se ocupa del análisis de algoritmos de *AA*. Se ha de tener en cuenta que los conjuntos de entrenamiento son siempre finitos, por lo que podremos cuantificar el error de generalización.

Dentro de este campo, encontramos los dos tipos de aprendizaje fundamentales, cada uno con sus ventajas y desventajas: aprendizaje no supervisado (Anexo 6.1), en el que se crea un modelo matemático sin tener ningún dato a priori, y supervisado.

2.2.1. Aprendizaje no supervisado

Los algoritmos de esta área reciben **únicamente** conjunto de datos, tener etiquetas: no le decimos al algoritmo qué es ese objeto, sino que él lo tiene que *adivinar*. Encuentran puntos en común, y aprenden según la presencia o ausencia de dichos puntos en cada nuevo dato. La búsqueda de relaciones se puede hacer mediante métodos algorítmicos diferentes.

Entre diversos, el ejemplo más conocido es el análisis de conglomerados o *clustering*. Este consiste en agrupar un conjunto de objetos de modo que los que estén en el mismo grupo sean más similares entre sí que a los de otros grupos. El algoritmo que se sigue es simple, se escogen unos puntos iniciales (las etiquetas iniciales) y los puntos que estén más cercanos a ellos tendrán la misma etiqueta. Se repite exhaustivamente

con puntos iniciales diferentes hasta que los patrones se repitan, entonces la máquina escoge el patrón más repetido.

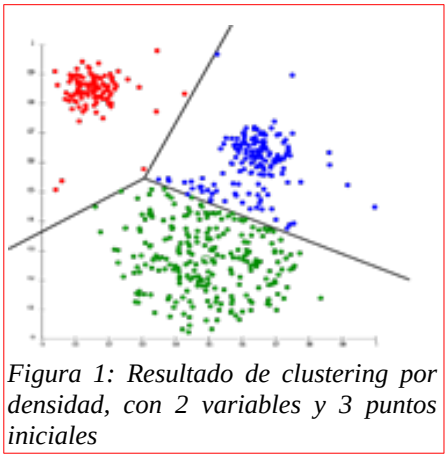


Figura 1: Resultado de clustering por densidad, con 2 variables y 3 puntos iniciales

2.2.2. Aprendizaje supervisado

En este tipo de aprendizaje el algoritmo también crea un modelo matemático, pero aquí el conjunto de datos entrantes tiene también los resultados deseados (etiquetas). Entonces se introduce un nuevo factor, el entrenamiento, que consiste en cómo el algoritmo procesa los datos iniciales para crear el modelo. Por ejemplo, si se quisiese identificar la presencia de un objeto en una imagen, se le darían imágenes al algoritmo, con las etiquetas correspondientes de si ese objeto está o no.

El objetivo es aprender una función utilizada para predecir la salida de entradas nuevas, a partir de ir optimizando iterativamente una función inicial. Dentro de este apartado, normalmente siempre nos referimos a dos tipos de algoritmos: clasificación y regresión.

- La regresión intenta estimar las relaciones entre una variable independiente y otra dependiente mediante procesos estadísticos. La función de regresión se puede entender como la recta que se encuentra justo en el medio de todos los puntos, para así predecir dónde se pueden situar los nuevos.

- La clasificación estadística consiste en el problema de identificar a cuál categoría pertenece un nuevo objeto, en base a unos datos de entrenamiento. Por tanto, los conjuntos de variables posibles pueden ser categóricos (por ejemplo: gato, perro, oso… para la especie de mamífero) u ordinales (por ejemplo: poco, regular, mucho y excesivo). Dado que este método está muy relacionado con el procedimiento práctico del trabajo, veamos dos algoritmos que llevan a cabo este proceso, de maneras totalmente distintas:

  ÁRBOL DE DECISIÓN: Un elemento con unas características conocidas ha de ser clasificado finalmente en una de las etiquetas de clase según las condiciones que cumpla (las ramas por las que pase). Son fáciles de interpretar y graficar, pero por otro lado un cambio en los datos para el entrenamiento puede hacer un gran cambio en las predicciones finales. Por ejemplo, si quisiésemos hacer un clasificador de correos spam o no-spam, deberíamos establecer unos patrones, tales como: si aparece la palabra “oferta”, hay errores ortográficos y el remitente no está en la lista de contactos, consideraremos como spam el correo. Contamos con unos datos de entrenamiento arbitrarios:

Etiqueta inicial	nº veces "oferta"	nº errores
Spam	4	4
No-spam	1	1
No-spam	0	2
No-spam	2	1
Spam	5	6
No-spam	0	2
Spam	3	4
No-spam	0	1
Spam	2	4
Spam	3	2

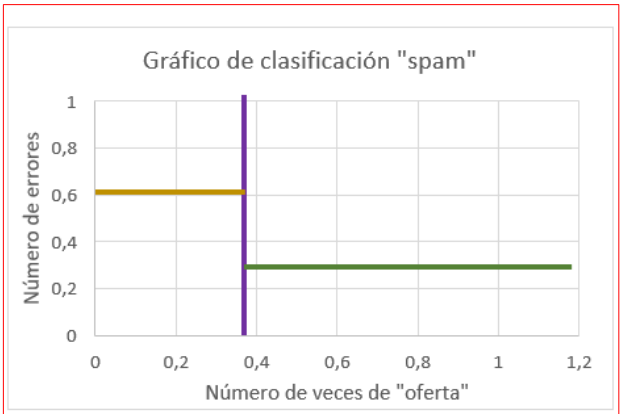


Figura 2 Gráfico de clasificación.

Como podemos ver, procesamos los datos y luego dividimos el gráfico con 3 líneas intentando separar lo más posible los correos de spam de los de no spam. Comenzamos estableciendo el límite en 2 veces la palabra “oferta”, y luego un número de errores correspondiente a cada subsección (una a la izquierda del 2 y la otra a la derecha). Nos queda el siguiente árbol:

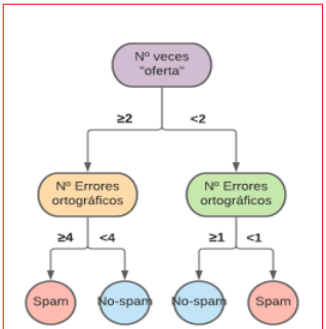


Figura 3 Diagrama de árbol

PERCEPTRÓN: Este es un algoritmo que actúa como un clasificador lineal: hace predicciones en base a una función lineal. El perceptrón recibe unas variables entrantes (características) x_1, x_2... x_n, y a cada una le corresponde un peso w_1, w_2... w_n que indica cuán influyente o importante es esa variable - el peso no deja de ser un número: 0,1 si la variable no importa, o 0,9 si importa. Inicialmente son aleatorios y con el entrenamiento, se van ajustando a un valor ideal. También tenemos una variable ***b*** denominada sesgo; esta nos ayuda a equilibrar la relación entre las entradas y pesos, para encontrar el patrón idea. Lo que hace el algoritmo es multiplicar el vector conformado por las variables entrantes y el vector de los pesos, sumando el sesgo, así obteniendo un valor y.

$$y = f\left[b + \sum_{i=0}^{n} w_i \cdot x_i\right]$$

La función f que estamos aplicando es la *función de activación*, que simplemente mantiene el conjunto de valores de salida en un rango de (0,1) o (-1, 1) para establecer un mínimo y un máximo que es fácilmente procesable por el ordenador. Normalmente la función Sigmoide se encarga de esta función.

Recordamos que la función Sigmoide tiene la forma: $f(x)=\frac{1}{1+e^{-x}}$. Podemos ver su forma en la siguiente figura, junto con la estructura general del clasificador:

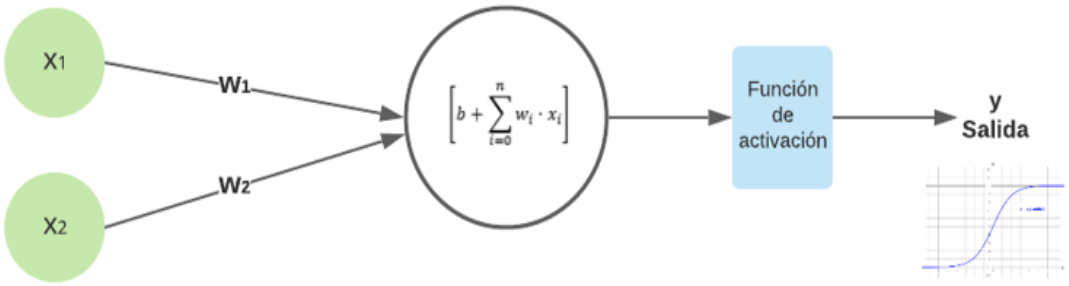


Figura 4 Componentes de un perceptrón simple. La y es la predicción del perceptrón (creación propia)

Si quisiéramos graficar el ejemplo usado en el apartado 2.1.5a, podríamos establecer una recta que separe las variables de entrada en dos grupos (spam y no-spam):

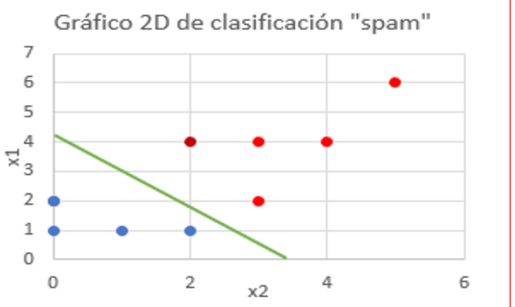


Figura 5: Hay una recta determinada que cumple con la clasificación

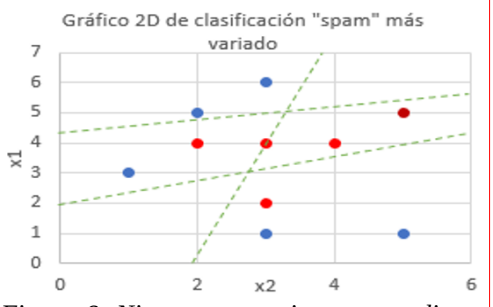


Figura 6: Ninguna recta tiene un pendiente adecuado.

Realmente, este gráfico debería tener 3 dimensiones, siendo la tercera dimensión el valor de y, el resultado de f. Solo hemos representado las dimensiones x_1y x_2 para facilitar la comprensión del problema. Entonces, el plano (la recta verde en la figura 3) es definido según la misma función y = $w_1 x_1$+$w_2 x_2$ + b, con unos pesos w_1y w_2 determinados que actúan como los pendientes de una función afín. ¿Pero qué pasaría si los datos tuvieran más variabilidad, como en la figura 4? No podríamos trazar una recta (por consiguiente, tampoco un plano) para separar los puntos, ya que calcule la que se calcule, tendrá un error demasiado grande. Así, la limitación del perceptrón simple es que **solo puede procesar conjuntos de datos linealmente separables.** La situación que me propuse tiene parámetros que obviamente no seguirán un comportamiento lineal, sino más bien complejo. Por esta razón, necesitamos un método que pueda separar diferentes variables de forma no-lineal. ¿Podría una red neuronal ser una solución para nuestra situación entonces?

2.3. Redes neuronales artificiales (RNA)

Las redes neuronales artificiales se inspiran en las biológicas, pero con el objetivo de resolver problemas de IA. Gracias a la conexión de nuestras redes neuronales, nuestros cerebros pueden procesar la información que le llega.

Si analizamos una sola neurona biológica, vemos que recibe información (en forma de impulsos eléctricos) gracias a las *dendritas*; lo característico de estos sistemas es la cantidad de neuronas conectadas entre sí, mediante conexiones llamadas *sinapsis*. Esta información es procesada en el núcleo (o *soma*), y es finalmente devuelta mediante el axón.

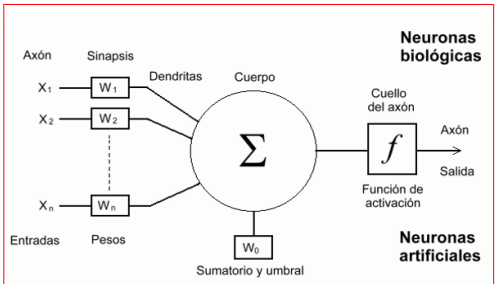


Figura 7: Concepto de la estructura de una neurona biológica y artificial.

Tomando este concepto, las dendritas de la neurona corresponderían con el vector de características o *inputs* de entrada; las sinapsis serían los pesos; el núcleo sería el sumatorio (o la operación que se lleva a cabo) y el cuello del axón correspondería con la función de activación. Aquí podemos ver claramente cómo el perceptrón simple sigue esta estructura.

Se trata de una red de neuronas (en este caso, con neurona nos referimos a un nodo que lleva a cabo operaciones matemáticas) organizadas en capas. Distinguiremos entre 3 tipos de capa principales: capa de entrada, capa oculta y capa de salida.

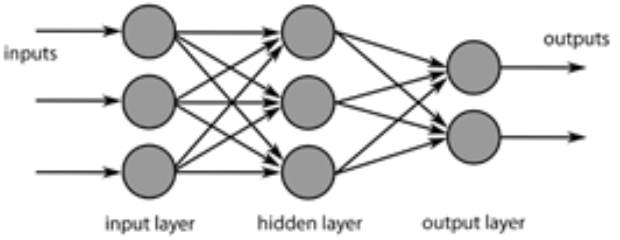


Figura 8: Red Neuronal Artificial con 3 inputs, una hidden layer con 3 neuronas y 2 outputs.

En efecto, cada neurona es un perceptrón, en la red neuronal que estudiaremos. Denominamos *feedforward* (o *forward propagating)* al proceso que hemos explicado del perceptrón: Las neuronas de la primera capa no realizan ninguna operación (transmiten directamente las entradas), sino que cada una de la capa oculta realiza: $b+\sum_{i=0}^{n} w_i \cdot x_i$. Se recorren todas las capas de la red, así acabando con tantos resultados como neuronas haya en la capa de salida (si queremos distinguir entre perro y gato, habrá únicamente 2 neuronas en esta capa). Gracias al resultado que dan todas las neuronas (diferentes rectas), podemos clasificar cualquier sistema no lineal, porque tendríamos que aumentar o disminuir el número de estas.

¿Qué es lo que hacemos para que los pesos lleguen a un valor ideal? Como estamos trabajando mediante un aprendizaje supervisado, tenemos unos datos de entrenamiento: datos de entrada de los cuales sabemos la salida correcta (z_i). De esta manera, podemos calcular el error, que queremos reducir a 0, utilizando **el error cuadrático medio** (Mean Square Error): una resta de “lo que obtenemos” menos “lo que esperamos”.:

$$\frac{1}{2}\sum_{i=1}^{n}(y_j - z_j)^2$$

La herramienta con la que calculamos los pesos se llama ***Backpropagation***. El procedimiento es: sabiendo el error que tenemos en la última capa, calculamos el ritmo de cambio de error que ha habido, y lo transmitimos a la capa anterior. Las neuronas que hayan contribuido más al cálculo del error que les han transmitido (tendrán un valor de función de activación mayor), reciben una mayor parte del error, y lo vuelven a transmitir a la

capa anterior. Esto se hace iterativamente según el número de capas, hasta llegar a la primera. Para entonces, se habrán cambiado todos los pesos (con diferentes valores para cada uno), que en un principio valían 0.

Calcular el ritmo con el que ha cambiado el error no es más que la derivada del error de los pesos y el sesgo, con respecto a las variables. ¿Por qué? Fijémonos en el siguiente gráfico de 3 dimensiones: $\theta_0 y \theta_1$ son dos variables (por ejemplo, el número de errores ortográficos y el número de veces que aparece “oferta”) y $J(\theta_0, \theta_1)$ es el error que ha habido en la red:

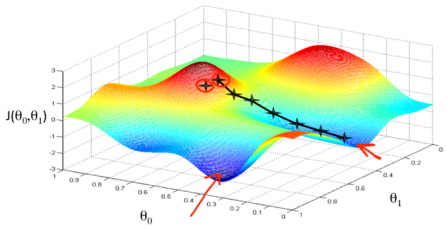


Figura 9: Representación del descenso por gradiente

Lo que buscamos, es llegar a un punto en el que el error sea lo menor posible: por esto **descendemos** por el gráfico. Esto lo hacemos calculando la derivada parcial de cada variable en un punto, que conforman el vector **gradiente**: la dirección de este nos indica hacia dónde la pendiente asciende. Si escogemos el sentido opuesto, podemos ir calculando la ruta por la cual descender.

Otra manera de verlo es la siguiente, pero esta vez en 2 dimensiones, con una sola variable ***w***:

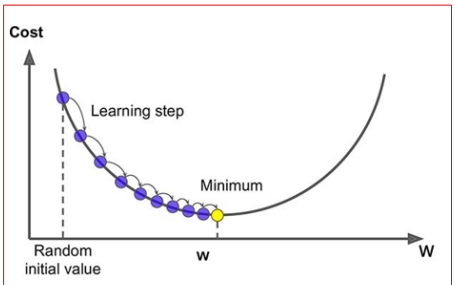


Figura 10: Descenso por el gradiente en 2D

2.4. La esencia del backpropagation

Recordemos que la derivada $\frac{df}{dt}$ nos devuelve el pendiente de la recta tangente a un punto: el ritmo con el que este cambia justo en ese momento. Daremos las reglas de derivación por supuestas, ya que no entra en el contexto del trabajo.

La derivada parcial de una función con más variables es la derivada respecto una de ellas, mientras las otras permanecen constantes:

$$f(x,y)=y^4+5xy$$

$$\frac{\partial f}{\partial x}=5y \qquad\qquad \frac{\partial f}{\partial x}=4y^3+5x$$

Lo último que necesitamos saber es la regla de la cadena: el proceso que nos ayuda a calcular derivadas de funciones compuestas. Si tenemos una función g(x) que depende de f(x):

$$\frac{d}{dx}[g(f(x))]=g'(f(x))\cdot f'(x)$$

Podemos entender una red neuronal como una función masivamente **compuesta**: cada capa de la red puede ser representada como una función.

Entonces ***backpropagation*** lo que hace es combinar estos dos conceptos (red neuronal y regla de la cadena de derivadas parciales): si queremos saber el ritmo de error E con respecto a la capa de pesos w_{ij}, tenemos primero que hacer la derivada parcial de E respecto a el cálculo que hace el perceptrón, luego la derivada parcial de E respecto la función de activación en la capa i. Esto, ya nos conduce a al cálculo del ritmo de cambio de error de E.

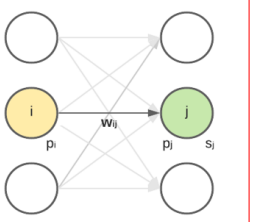


Figura 11: p_i y p_j son el cálculo que hace el perceptrón

$$\frac{\partial E}{\partial p_j}=\frac{\partial E}{\partial y_j}\cdot\frac{\partial y_j}{\partial p_j}=\frac{\partial E}{\partial y_j}\,y_j(1-y_j) \qquad (1)$$

Recordemos que $\frac{\partial y_j}{\partial p_j}$ es la derivada de la función sigmoide. Procedemos.

$$\frac{\partial E}{\partial y_i}=\sum_j \frac{\partial E}{\partial p_j}\cdot\frac{\partial p_j}{\partial y_i}=\sum_j \frac{\partial E}{\partial p_j}\,w_{ij} \qquad (2)$$

Consideremos que $p_j=\sum_i (¿y_i\cdot w_{ij})¿$

$$\frac{\partial E}{\partial w_{ij}}=\frac{\partial E}{\partial p_j}\cdot\frac{\partial p_j}{\partial w_{ij}}=\frac{\partial E}{\partial p_j}\,y_i \qquad (3)$$

De la primera ecuación (1), tenemos que

$$\frac{\partial E}{\partial w_{ij}}=\frac{\partial E}{\partial y_j}\,y_j(1-y_j)\,y_i$$

Si E es el error cuadrático medio: $E=\frac{1}{2}\sum_j (¿z_j-y_j)^2¿$. Entonces:

$$\frac{\partial E}{\partial y_j}=-(z_j-y_j)$$

$$\frac{\partial E}{\partial w_{ij}} = -y_i y_j (1-y_j)(z_j - y_j)$$

Con esta fórmula, sabemos el ritmo del cambio, que podremos aplicar a **cada peso** por cada época del programa: cuando se va yendo a través de las capas de la red. Si el peso tiene que cambiar, le restaremos el cambio al propio valor, ya que no podemos asegurar que un cambio lineal o exponencial vaya a ser eficaz.

$$w_{ij} = w_{ij} - \eta \frac{\partial E}{\partial w_{ij}}$$

Hay que destacar que η es el ritmo con el cual se va descendiendo por el gradiente: si es muy grande la precisión será poca y seguramente tengamos error. Si es muy bajo, tardaríamos mucho en computar el cálculo. Encontrar su valor ideal es una cuestión muy investigada hasta el día de hoy, con diversas propuestas. Nosotros, como veremos en la parte práctica, utilizaremos un valor determinado.

En conclusión, es la manera en que encontramos nuestros pesos ideales: entrenamos nuestra red neuronal con datos de entrenamiento, obteniendo unos resultados y su **error correspondiente**. Utilizando ***backpropagation***, usamos la derivada parcial de este error en base a todas las anteriores capas (la última función de cálculo de error, depende de: la función de activación y el cálculo del sesgo más los pesos multiplicados por las entradas).

2.5. Aprendizaje profundo (AP)

Explorando la otra rama en el aprendizaje, el aprendizaje de carácter profundo busca lo mismo que el automático: tener la habilidad de aprender patrones que no sean explícitos. Pero va más allá, intentando procesar muchos más datos. La esencia del *AP* está inspirada en la

estructura del cerebro humano, en el sentido de imitar la **enorme** cantidad de conexiones neuronales que se usan para tratar la información. Por tanto, el *AP* se centra principalmente en usar redes neuronales **profundas**, con capas de centenares o miles de neuronas. Aumentamos el nivel de abstracción: esta es la única diferencia a nivel teórico respecto el *AA* tradicional. Así como se hacen cálculos más extensos, necesitamos un mayor volumen de datos, lo cual implica más poder de cómputo y tiempo.

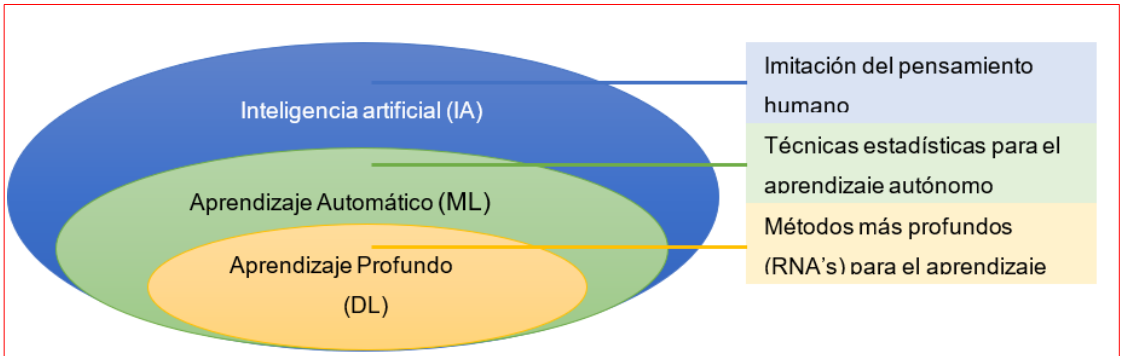


Figura 12: Esquema de los tipos de aprendizaje que hemos visto (creación propia).

Una vez vistas las bases del trabajo práctico, los diferentes modos en que una máquina inteligente puede desarrollar un proceso, podemos contextualizar la materia en la que se fundamenta: el reconocimiento de objetos.

2.6. Visión por computador (VC)

Comenzamos contextualizando la tan compleja detección de fenómenos. La visión por computador es un campo encargado de comprender el contenido de una imagen o vídeo. Se incluyen los métodos de adquisición y procesamiento de imágenes digitales, para que una vez con los datos, se puedan tomar decisiones. La *IA* y la *VC* van casi siempre juntas: la *VC*

necesita *IA* a fin de imitar el sistema visual humano. De todos los campos que contiene, tales como la fotogrametría, modelado de objetos y el seguimiento de eventos, nos centraremos en el reconocimiento de objetos.

2.6.1. Reconocimiento de objetos

Este es el proceso de encontrar e identificar concretamente objetos de cualquier tipo en una matriz de píxeles. Veamos primero en qué consisten los métodos tradicionales para la clasificación, en el reconocimiento de objetos.

2.7. Métodos tradicionales para la clasificación en el reconocimiento de objetos

A los humanos, la variación ligera del punto de vista, tamaño, translación o rotación no nos afecta mucho al identificar una entidad, incluso cuando no está totalmente visible. Sin embargo, para los ordenadores esto es todo un desafío. El concepto objeto lo denotamos como una entidad del mundo real con una etiqueta única.
Se busca encontrar una manera de clasificar estos objetos. Hay diferentes métodos tradicionales para hacer esto: el análisis de características (de funcionamiento muy similar al *clustering*) o la comparación de plantillas.

Comenzamos con el análisis de características. Este se fija en la tipología de unas características determinadas. Dependiendo de en qué aspectos nos fijemos, los patrones que se encuentren se asignarán a una clase u otra, teniendo un cierto grado de similitud. Los tipos de características pueden ser: topológicas (p.e.: número de agujeros, número de componentes cóncavas), geométricas (p.e: área, perímetro) y estadísticas (momentos). Si quisiéramos diferenciar tres tipos de flores

(*Iris virginica*, *Iris versicolor* e *Iris setosa*) según la anchura de sus pétalos (x_1) y su longitud (x_2), almacenando el patrón en un vector x= [x_1, x_2], podríamos filtrar los patrones de dos tipos de flor, pero no de los tres:

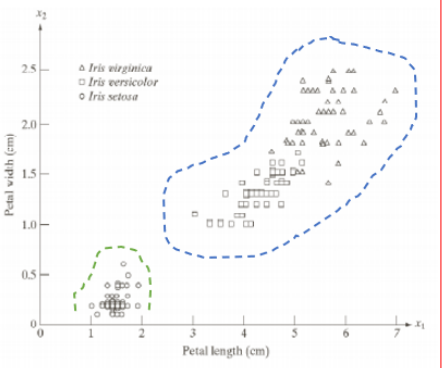


Figura 13: Gráfico de la representación de 3 flores según dos parámetros X_1 y X_2

Para que este método diese buenos resultados, tendríamos que especificar con mucho rigor las características de los elementos. Cuando se trata de contadas propiedades, esto no es un problema. Pero cuando queremos estudiar objetos como por ejemplo un insecto, necesitamos más de 2 propiedades (2 dimensiones), haciendo difícil la extracción de resultados.

La comparación de plantillas es otro método, en que se compara la imagen en cuestión a un patrón visual mediante una plantilla almacenada, y determina si encaja o no. Este procedimiento sería efectivo si la forma básica es constante, pero si hay una ligera variación ya sea en el tamaño, posición o grado de perspectiva, el reconocimiento resulta mucho más complejo, requiriendo mucho tiempo de cómputo. Lo mismo pasa con el método de descripción de objetos, el cual requeriría un alto grado de especificación.

En vista de ello, cuando las características a analizar son numerosas y los métodos tradicionales no nos sirven, entra en juego el *AA*, en concreto el

AP. Las *RNA* en el *AP* hacen que la máquina aprenda casi por si sola, no le tenemos que especificar las características a detectar, sino que el algoritmo las deduce según los datos de entrenamiento (aprendizaje supervisado). Por esto el *AP* se utiliza en tareas complejas como lo es esta, el reconocimiento de objetos. Procederemos entonces a explicar en qué consiste mi propuesta.

3. Parte práctica: Funcionamiento de una CNN

3.1. Las redes convolucionales (CNN)

Teniendo el fundamento necesario sobre el funcionamiento de una red neuronal multicapa y el cálculo de sus pesos, puedo proceder con fluidez a la explicación del método que me ha parecido que más satisfactoriamente lleva a cabo un reconocimiento de objetos.

Una Red Neuronal Convolucional (*Convolutional Neural Network*) es un tipo de *RNA*, basándose en aprendizaje supervisado, en la que sus capas imitan la función del córtex visual humano: identificar las características más eficientes de un objeto. Con "eficiente" hago una diferencia entre características ambiguas y útiles: distinguimos más rápido dos tipos de vehículos por su forma que por su color; en cambio, distinguimos más rápido dos tipos de flor por su color que por su forma.

Para esto, la CNN tiene varias capas ocultas que siguen una **jerarquía**: las primeras detectan líneas y curvas, las siguientes figuras geométricas, los siguientes patrones generales, así hasta reconocer formas complejas como una silueta.

Antes que nada, ¿por qué no usamos una red neuronal común, como la vista anteriormente? Porque una red neuronal tradicional tendría que procesar 224x224x3x1024 = más de 150 millones de pesos para la primera capa, si el objeto fuese a color y con resolución 224x224 y. En

vez de malgastar energía de cómputo procesando cada neurona como a cada píxel de la imagen, nos fijaremos en el contexto del píxel: sus vecinos.

3.1.1. Imágenes

La materia que trabajaremos serán las imágenes. ¿Cómo las trata un ordenador? Una imagen puede ser procesada como una **matriz de píxeles**, como si fuera una tabla gigante de n columnas y m filas, en la que cada celda es un píxel. Si únicamente trabajáramos con una tonalidad (blanco-gris-negro), el valor que podría almacenar cada celda iría de 0 a 255. Si tuviéramos una imagen a color, necesitamos los 3 canales RGB (red, green, blue), y esta matriz pasaría a tener 3 dimensiones, tal como vimos antes.

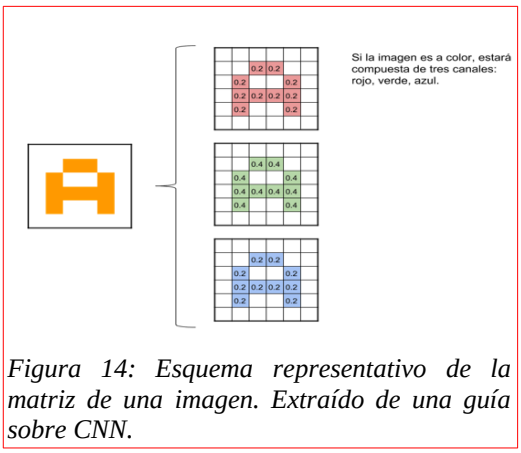


Figura 14: Esquema representativo de la matriz de una imagen. Extraído de una guía sobre CNN.

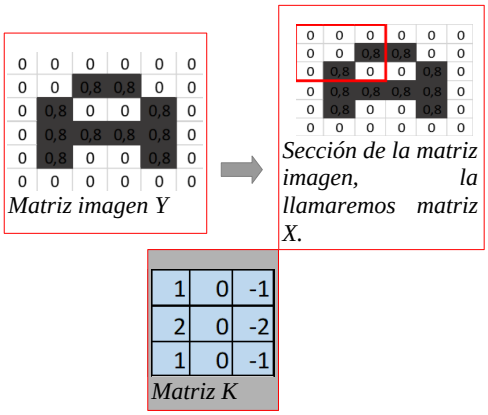
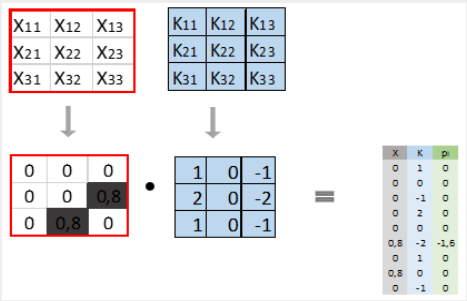
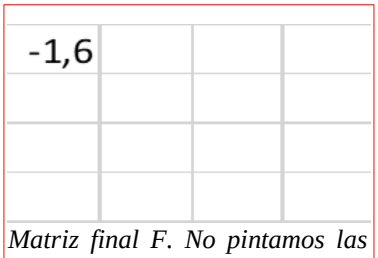
Transformaremos el valor de todos los píxeles reduciéndolo de 0 a 1 (dividimos por 255), ya que la red neuronal procesa este rango con más fluidez.

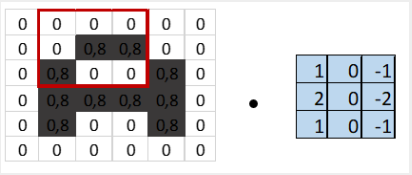
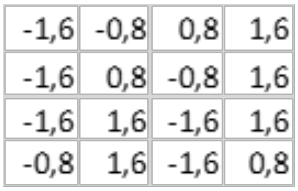
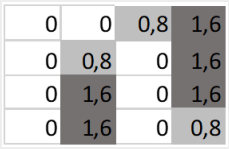
En una red convolucional, las capas realizan la **convolución:** se toman "grupos de píxeles cercanos" contenidos en la imagen de entrada, y se combinan (producto escalar) con una pequeña y determinada matriz de pesos (también conocido como *kernel* o filtro, y se obtiene mediante

backpropagation). Así, resulta una matriz final F, en la que cada elemento de esta corresponde con cada resultado de la multiplicación de las dos matrices anteriores (posteriormente veremos exactamente cómo). Si establecemos una analogía con el algoritmo del Perceptrón, entendemos como a variable de entrada x_1 una sección de la matriz de píxeles, y el peso $w_1$como la matriz de pesos. Así que podemos deducir la función de este último: tener unos valores ideales para que, al multiplicarlos por la matriz de entrada, nos de otra matriz resultante que nos sea útil para reconocer **características**.

3.1.2. Primera convolución

Profundicemos en el cálculo que se hace. Supongamos que tenemos una matriz imagen Y, que representa la letra A. Tenemos una matriz $kernel\ K$ con dimensiones 3x3 en este caso (el $kernel$ suele tener el mismo alto que ancho), con unos valores escogidos arbitrariamente. En los apartados posteriores veremos que el objetivo de las CNN es justamente encontrar estos valores mediante el entrenamiento.

Paso	Descripción	Soporte gráfico
1	Se hace un producto escalar entre la matriz *X* y *K*: la suma de multiplicar cada elemento de una matriz con el correspondiente de la otra. Es decir, multiplicamos el primer elemento X_{11} con el primer elemento K_{11}; guardamos en el vector *p*. Seguimos: multiplicamos el segundo elemento X_{12} on el segundo elemento X_{12} y lo guardamos en p_2. Así sucesivamente, hasta tener 9 valores.	
2	*Se hace un producto escalar entre la matriz X y K*: la suma de multiplicar cada elemento de una matriz con el correspondiente de la otra. Es decir, multiplicamos el primer elemento X_{11} con el primer elemento K_{11}; guardamos en el vector *p*. Seguimos: multiplicamos el segundo elemento X_{12} con el segundo elemento X_{12} y lo guardamos en p_2. Así sucesivamente, hasta tener 9 valores.	
3	*Se suman todos los valores que resultan del producto escalar*: $p=p_1+p_2+p_3+\dots p_n$. $p=0+0+0+0+0+(-1{,}6)+0+0+0=-1{,}6$ Esta suma será el **valor del píxel** correspondiente a la **imagen final.** Es decir, que el resultado corresponderá con el elemento F_{11} de la matriz final *F*.	 *Matriz final F. No pintamos las celdas porque falta aplicar la función de activación.*

4	*Ahora volvemos al paso 1. Escogemos la siguiente sección X* de la matriz *Y*, que será avanzar una columna. De hecho, siguiendo este orden podemos deducir las dimensiones de la matriz *F*: solo podemos avanzar 4 veces horizontalmente (columnas), y 4 veces verticalmente (filas), así que las dimensiones serán 4x4.	
5	*Volvemos a hacer los pasos 2 y 3. El resultado lo almacenamos en* F_{12} . Si hacemos estos 5 pasos iterativamente, recorriendo toda la matriz Y, habremos hecho 16 productos escalares, y tendremos nuestra matriz F completa. Aunque en este ejemplo se reduzca las dimensiones de la imagen (de 6x6 a 4x4), hemos de considerar que siempre se utilizan métodos para que las dimensiones no varíen (agrandando la imagen inicial con píxeles vacíos, y luego obteniendo una matriz de las dimensiones queridas).	
6	*Ahora volvemos al paso 1. Escogemos la siguiente sección X de la matriz Y,* que será avanzar una columna. De hecho, siguiendo este orden podemos deducir las dimensiones de la matriz *F*: solo podemos avanzar 4 veces horizontalmente (columnas), y 4 veces verticalmente (filas), así que las dimensiones serán 4x4.	$f(x) = \begin{cases} 0 \text{ si } x < 0 \\ x \text{ si } x \geq 0 \end{cases}$ 

	Hemos filtrado nuestra imagen inicial con el *kernel*, obteniendo otra imagen que tiene rasgos similares. De hecho, no solo podemos aplicamos e1 filtro, sino que en una misma capa de la red convolucional puedo aplicar más de un filtro, así teniendo tantas imágenes resultantes como filtros aplique.	

Pero, ¿con qué objetivo hacemos esto? Hemos llevado a cabo un ejemplo simple, pero veamos qué pasaría si trabajáramos con una imagen grande, pero con un *kernel* determinado, llamado *vertical Sobel filter*:

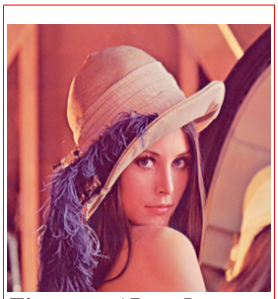


Figura 15: Imagen, como si fuera nuestra matriz Y

-1	0	1
-2	0	2
-1	0	1

Figura 16: Imagen final, como si fuera nuestra matriz F

Si aplicáramos el *horizontal Sobel filter:*

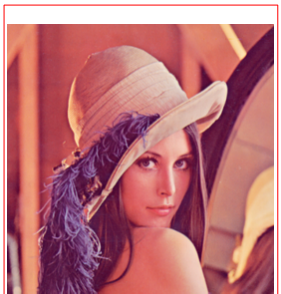


Figura 17: Imagen matriz Y

1	2	1
0	0	0
-1	-2	-1

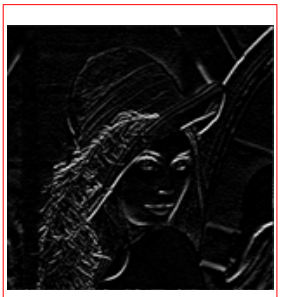


Figura 18: Imagen matriz F

Como vemos, estos filtros (*kernels*) son detectores de bordes: el vertical detecta bordes verticales, y el horizontal, bordes horizontales. Así, las características de la imagen quedan reveladas, haciendo fácil a la red neuronal convolucional detectar patrones que cumplen ciertos objetos. A modo de ejemplo, una boca tendrá en común muchos bordes horizontales, con lo que este filtro le será útil a la máquina.

3.2. Subsampling - maxpooling

La cantidad de neuronas que estaríamos procesando sería enorme. Si tuviéramos una imagen con blancos y negros de 28x28 píxeles, tendríamos una capa de entrada compuesta de 784 neuronas. Con la primera convolución, si utilizáramos 32 filtros, acabaríamos con 25088 neuronas para la siguiente capa (32 imágenes finales con 784 neuronas cada una).

Queremos reducirlo, pero las características importantes de cada filtro deben quedar visibles, a esto lo llamamos *subsampling*. Dentro de este campo, utilizaremos método del *max-pooling*. El *max-pooling* consiste en aplicar un filtro de una manera especial. Analizaremos secciones de la imagen de 2x2 (esto equivale a decir que la *pool* es de 2), y de esos 4 elementos, escogeremos el mayor (de aquí viene el prefijo *maX*).

Lo guardamos en el elemento correspondiente de la matriz final. Luego avanzamos de dos en dos unidades, no de una en una como hacíamos con el ***backpropagation*** y hacemos lo mismo con las otras secciones. Por ejemplo:

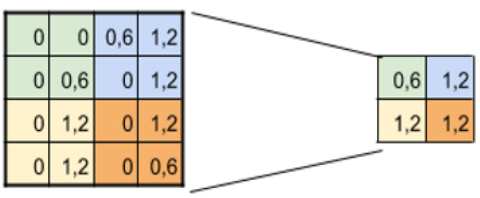


Figura 19: Ejemplo de max-pooling, reduciendo la imagen a la mitad

De esta manera, con el ejemplo de las imágenes de 28x28 píxeles, las reduciría a 14x14, y aplicando los 32 *kernels*, me quedarían en 6272 neuronas.

3.3. Siguientes convoluciones

Recordemos que podemos hacer tantas convoluciones como queramos, siempre y cuando lleguemos a unas dimensiones pequeñas de las imágenes finales.

Podemos decir que hemos conformado una convolución, que consta de una entrada, una serie de filtros que corresponde con una serie de imágenes finales (conocidas como *mapas de características*, ya que justamente destacan lo más característico) y luego reducimos las dimensiones de la imagen mediante *max-pooling*. Es decir, resulta una estructura como la que refleja la figura 20.

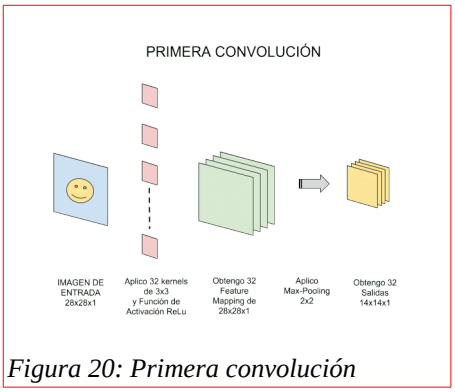


Figura 20: Primera convolución

La segunda convolución seguramente sea capaz de extraer características más profundas que líneas o curvas. Su entrada de datos serán los datos de salida la convolución anterior: 32 imágenes de 14x14.

Arbitrariamente, aplicaré 64 *kernels* a todas estas imágenes, así teniendo *kernels* de dimensiones de 3x3x32, aplicando también la función ReLU para descartar los valores negativos. La salida, serán 64 imágenes con las características, a las cuales aplicaré *max-pooling* de 2x2. Obtendré 64 imágenes reducidas a la mitad, de 7x7.

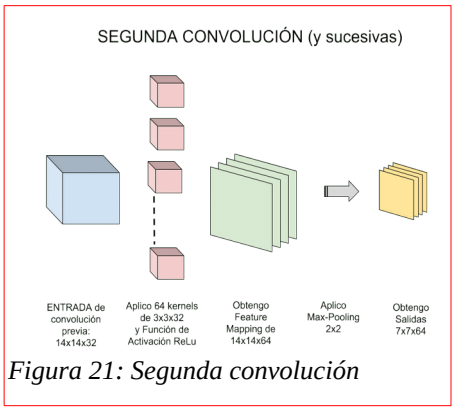


Figura 21: Segunda convolución

Añadiremos una tercera convolución, ya que las imágenes siguen teniendo aún unas dimensiones grandes. Aplico 128 filtros de 3x3, obteniendo 128 imágenes, así como también el mismo *max-pooling*; pero esta vez, las imágenes se reducirán a 3x3. Como las dimensiones ahora ya son pequeñas, dejamos de añadir capas convolucionales y convertimos la última salida (128 imágenes de 3x3) en una red neuronal.

3.3.1. Transformación a red neuronal tradicional

Como hemos dicho, haremos una red neuronal (conformada por perceptrones) en que cada neurona corresponderá con cada píxel de las imágenes finales, que contendrá las características más elaboradas.

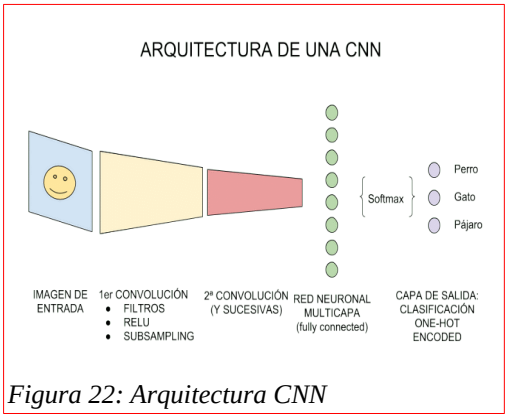


Figura 22: Arquitectura CNN

Es decir, mientras que, en las primeras capas de la red convolucional, las neuronas hacían referencia hacia si había una línea o no, en las últimas capas las neuronas nos dirán directamente la probabilidad **con que se cumplen un conjunto de características** que son las características de un objeto especificado en el entrenamiento. Una vez tenemos la capa de neuronas tradicionales, le aplicamos a esta capa de neuronas la función *Softmax*: recibe todos los valores que le dan las neuronas de las capas anteriores, los multiplica, y pasa el valor de cada una a una probabilidad de 0 a 1 (rango definido). Luego, se pasa directamente a la capa de neuronas con las clases a clasificar. Si clasificáramos entre el carácter A y B, tendríamos 2 neuronas. De manera que, si se calcula que una imagen cualquiera cumple con las características de la A, podría haber la siguiente distribución probabilística: [0.6, 0.4]. Como el valor de la A (calculado por *Softmax)* es mayor, se retornaría un valor de [1,0].

¿Y cómo se calculan los valores de las matrices de *kernels*, que son los que realmente determinan las características?

Entrenando nuestra red mediante aprendizaje supervisado, lo que hacemos es utilizar el descenso de gradiente. Así, se van ajustando los valores de los *kernels* (encontrar el valor deseado para el cual el gradiente = 0), que no son más que conjuntos de pesos. Por eso, no es necesario ver otra vez el cálculo diferencial que fundamenta esto (***backpropagation***), ya que es repetitivo. En el caso expuesto, deberíamos ajustar los 9 (3x3) parámetros de 32 filtros, lo cual es poco en comparación a los millones de pesos que deberíamos encontrar si comparáramos las imágenes píxel por píxel.En los Anexos 6.5 y 6.6 lo explico detalladamente. Y con el Anexo 6.6, entendemos que pasar de una matriz a la que le vamos modificando el valor del píxel para encontrar patrones a una red neuronal

que nos devolverá la etiqueta de la imagen, resulta de hacer todas las convoluciones. Comenzamos con muchos valores, y acabamos con tantos como etiquetas hayan.

3.4. Aplicación de la Teoría

El cálculo que conllevan las multiplicaciones matriciales (de las convoluciones y de la propagación del error) es enorme. Necesitaríamos un software únicamente dedicado estas tareas, y es justamente lo que utilizaré: TensorFlow.

Esta es una librería de código libre dedicada al Machine Learning (aprendizaje automático), a través de un rango de métodos. Ha sido desarrollado por Google con el objetivo de satisfacer sus propias necesidades de trabajar con redes neuronales artificiales. Mayoritariamente está basado en redes neuronales de aprendizaje profundo. De esta manera, podemos decir que es una herramienta de computación numérica. Podemos utilizar esta librería mediante lenguajes de programación como Python o C++.

Únicamente para aclarar el contexto informático en el que nos encontramos, hay que decir que un lenguaje de programación es una manera de comunicar a nuestro ordenador todas las instrucciones que queremos que haga: funciones iterativas (bucles), cálculos complejos, operaciones con matrices, crear o modificar archivos del propio sistema, acceder a servidores externos para obtener información, y un largo etcétera.

Entonces, esta librería contiene el procedimiento matemático que constatamos el marco teórico. Nos ayudará en el cálculo de pesos de las redes neuronales, teniendo que obtener el valor de las derivadas parciales correspondientes.

Tal como indica el nombre, se trabaja con tensores. Estos, no son más que vectores de n dimensiones. Cabe remarcar que, en la jerga

informática, un vector es un conjunto de identidades; como por ejemplo un conjunto de números, o de variables, o de letras, o de palabras... A modo de ejemplo:

- Vector de 0 dimensiones: escalar (número).
- Vector de 1 dimensión: un grupo de números.
- Vector de 2 dimensiones: una matriz (con filas y columnas)
- Vector de n dimensiones: se puede entender como un grupo de grupos de grupos de grupos... así sucesivamente.

3.5. Desarrollo de mi propuesta

Durante los últimos dos meses he estado probando diferentes estructuras de redes neuronales convolucionales. Hemos visto en el apartado 3.1 la estructura base de una red neuronal convolucional, pero esto claramente no nos garantiza la mayor eficiencia. A continuación, sabiendo cuáles son las piezas de las que se suelen componer estas redes (los tipos de capa: Convolución y *Maxpool*), investigaremos cuál es la combinación de estas que conformará una estructura que dé los mejores resultados posibles para clasificar números del 0 al 9. Antes, hay que dejar claros unos puntos:

¿Hay una única estructura correcta?	No. Recordemos que nos estamos centrando en una única situación, pero en la realidad hay muchas más, y las estructuras requeridas serán diferentes. Los objetos contenidos en una imagen pueden variar mucho, y según los patrones de forma o color de estos, hay unos modelos de CNN que serán más o menos convenientes.
¿Qué limita el rigor de la creación de este modelo?	Claramente esta es la parte del trabajo con menos rigor, ya que no podré probar todas las estructuras que quisiera, sino que me quedaré

	con la que mejores resultados me dé. Esto es debido a que no dispongo de un ordenador con mucha capacidad computacional. Nos centraremos más en obtener el mejor rendimiento, sin profundizar tanto en los conceptos teóricos, ya que ahora sí disponemos de una base sólida.
¿Cuáles son las variables que variaré, en las diferentes estructuras que construya y pruebe?	Principalmente son: • El número de capas de la red. • La función de activación (la que habíamos visto es la función ReLU, pero también se puede trabajar con la función Sigmoide, Tangente hiperbólica…). • El tamaño de la matriz del kernel (que habíamos asumido como a 3x3).

Esta estructura puede ser entendida también como un modelo (de red neuronal convolucional). Para valorar cuál modelo es mejor que otro, me basaré en los resultados del entrenamiento de cada modelo. Sin embargo, debemos entender primero en qué consiste el entrenamiento de los datos en esta situación particular.

3.6. El entrenamiento de la CNN

Una red neuronal profunda requiere de una muestra de datos **muy grande**, por lo que obviamente no podría crear una base de datos desde 0 (con unas inputs y outputs hechas manualmente), y esperar que la red me devuelva buenos resultados. Así que he recurrido a usar una base de datos externa, llamada MNIST. Esta contiene más de 60.000 imágenes de números del 1 al 9 (con sus etiquetas respectivas), que sirven como a datos de entrenamiento, y más de 10.000 imágenes para usar como a datos de validación. ¿Qué diferencia hay entre datos de entrenamiento y

de validación? Como el nombre indica, el primer conjunto de datos se utiliza cuando el algoritmo es totalmente nuevo: los pesos de las neuronas tienen un valor aleatorio (no estrictamente aleatorio, ya que esto es actualmente imposible de hacer en la informática, si no que es un valor cualquiera, irrelevante). Entonces, con estos datos de entrenamiento estos pesos se “calibran”, y cuando ya tienen un valor asignado, entonces ponemos a prueba al modelo con los datos de validación. Los dos grupos de datos son simplemente diferentes: si los datos de entrenamiento son imágenes de “cosas” que se parecen a perros, pues los de validación también serán “cosas” parecidas a perros, aunque diferentes ejemplos.

Estas imágenes tienen unas restricciones: son de 28x28 píxeles, y en blanco y negro. Como valoro más disponer de una amplia base de datos, y esta es una de las más populares en el mundo del reconocimiento de objetos, me limitaré a estas restricciones. Esto se verá a la hora de hacer el código de cada modelo.

Los datos de validación los usaremos para comprobar la eficiencia de los datos, con la ayuda de una tabla de convergencia. Está claro que como a resultado, tendré un modelo al que, pasándole una imagen de un dígito, me dirá cuál es, pudiendo acertar o no. Una tabla de convergencia simplemente nos ayuda a visualizar qué cantidad de dígitos ha acertado y fallado: podemos ver qué dígito ha sido confundido por qué dígito. Luego la veremos en detalle.

El entrenamiento, como hemos visto durante este trabajo, consistirá en ir procesando repetidas veces las imágenes de entrenamiento, e ir ajustando los pesos en función al resultado esperado. Pero esto se hace de una forma ordenada: vamos analizando los datos por lotes (*batch*) para no colapsar la red, y cuando se han recorrido todos los datos, decimos que se ha completado una época (*epoch*). Los lotes suelen ser definidos

(en nuestro caso serán de 128 datos en 128 datos), mientras que las épocas pueden variar. Durante las pruebas, incrementaremos el número de épocas tanto como mi ordenador lo permita (sin que se llegue a sobrecalentar).

Lo último que queda por explicar son la función de error y el optimizador. Las opciones (de ambos apartados) expuestas a continuación son extraídas directamente de Tensorflow. Su funcionamiento tiene un grado más de complejidad para ser explicado, pero siguen un método matemático muy similar al que ya hemos visto. La función de error es simplemente la manera en que calculamos las diferencias entre cada resultado. En el apartado 2 nos centramos únicamente en la función de error cuadrático medio, pero hay otros métodos, como la entropía cruzada categórica (nos es suficiente con saber que siguen el mismo objetivo, con definiciones diferentes). El optimizador es el método con el cual encuentro el valor de los pesos ideales. Exploramos y vimos cómo funcionaba el método por *gradient descent*, pero también hay alternativas que siguen el mismo concepto: Adam, Adagrad, RMSprop (solo trabajaremos con el primero). Entre ellos, se diferencian por la implementación de aceleración de aprendizaje, momento, y regresiones.

Como las imágenes no son muy grandes, no vale la pena variar el número de capas de la red: mejor mantendremos bajo. Por esta razón he adoptado un modelo básico, con 2 convoluciones y 2 capas de Max-pooling correspondientes. Las capas posteriores acaban de trabajar con el resultado de aplicar las convoluciones.

Capa	Propiedades
Convolución 1	6 kernels (filtros): 5x5 o 3x3. Activación: ReLU, Tanh o Sigmoide
Max-pooling 1	6 kernels (filtros): 2x2. Sin activación.
Convolución 2	16 kernels (filtros): 5x5 o 33. Activación: ReLU, Tanh o Sigmoide
Max-pooling 2	16 kernels (filtros): 2x2. Sin activación.
Flatten	Transforma la matriz en vector unidimensional
Conexión total	Conexión de 120 neuronas. Activación: ReLU, Tanh o Sigmoide
Conexión total	Conexión de 84 neuronas. Activación: ReLU, Tanh o Sigmoide
Salida	10 posibles salidas. Activación: Softmax

Procedemos a exponer los resultados de las pruebas. En total son 24 pruebas (dado que cambiamos los tipos de funciones, optimizadores y dimensión del *kernel*). Hemos decidido hacer un gran número de pruebas para estar completamente seguros del buen funcionamiento de cada variable en relación a las otras. Ya que, aunque una variable en concreto fuese poco eficiente individualmente, combinada con otra variable pueden tener un desempeño inigualable. Solo hemos puesto la mitad de las pruebas para no ocupar demasiado espacio, pero se pueden ver el resto en los Anexos del trabajo. Aquí ya utilizo uno de los archivos del código: *cnn.py*. Este archivo me permite generar una CNN con las variables que me interesan. Resultados en el Anexo 6.1.2.

Luego de realizar el entrenamiento, los parámetros (función de error, opimizador y dimensión del kernel) más adecuados han sido los de la prueba 19:

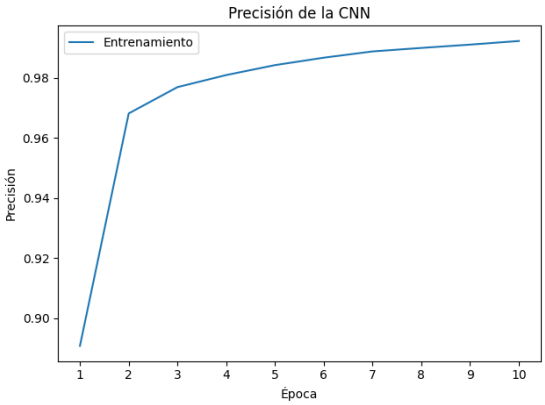


Figura 23: Gráfico de la evolución de la precisión del modelo a lo largo del tiempo (dividido en épocas), de creación propia

Prueba 19:

- Función de activación: ReLU
- Función de error: Entropía Cruzada Categórica
- Dimensiones *kernel*: 5x5
- Número de Épocas: 10
- Optimizador: Adam
- Precisión final: 99,40%

3.7. Construcción del modelo

Analizando las pruebas que hemos realizado (fijándonos en las pruebas de los anexos también), extraemos las siguientes conclusiones para el reconocimiento de dígitos numéricos:

- Las dimensiones del kernel no influyen mucho en el modelo.
- La parte que más determina la precisión del modelo es en definitiva el optimizador. Notamos que implementar Adam es una muy buena decisión para sacarle el máximo provecho a la CNN.
- Necesitamos una función de activación que delimite su rango en uno parecido a [-1 a 1], ya que si no, los valores dejan de seguir un patrón. Consideramos que ReLU es una de las mejores opciones.
- En cuanto a la función de error, ambas opciones dan buenos resultados, pero elegiremos la de Entropía Cruzada Categórica porque nos devuelve una mayor precisión, combinada con Adam y ReLU.

Teniendo esto claro, podemos construir nuestro modelo definitivo. Una representación gráfica nos ayuda a entender la estructura. Tengamos en cuenta las dimensiones que van cambiando. En las primeras capas, se va

incrementando la profundidad, esto es porque se hace una capa por cada filtro aplicado (si aplicamos 6 filtros, tendremos 6 unidades de profundidad). Pero esto solo nos sirve para hacernos una idea geométrica, ya que la red solo trabaja con vectores de dimensiones que varían.

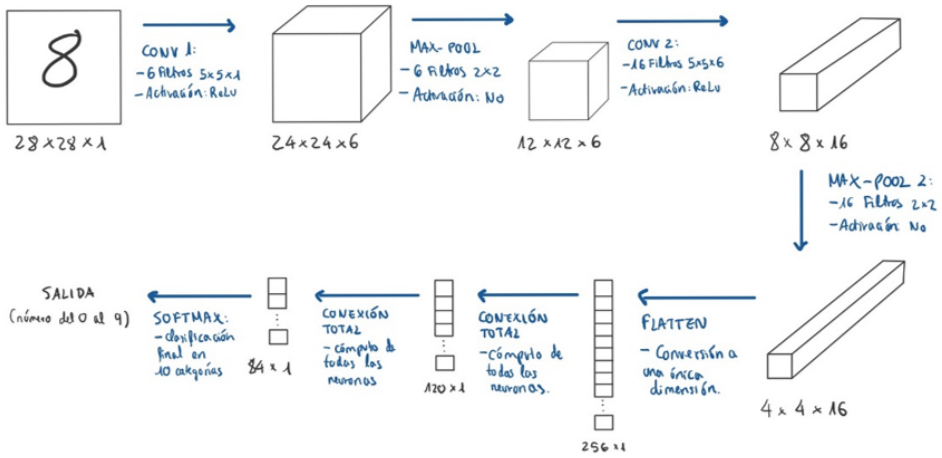


Figura 24: Mi propio modelo de CNN (creación propia)

Antes de probarla con dígitos escritos sobre papel (haciéndoles una foto con mi teléfono móvil), veamos cómo reacciona al procesar las imágenes de validación, de MNIST también. Ahora utilizo el archivo *evaluar_mnist.py*, cuya función tiene obtener la red neuronal convolucional generada anteriormente, importar las imágenes de la base de datos, y devolver la precisión con la que se ha acertado la etiqueta de las imágenes, así como una matriz de confusión.

Después de ejecutar el archivo, vemos que hemos obtenido una precisión del 98,58%. Recordemos que la precisión anterior de 99,40%, corresponde con datos de entrenamiento, no de validación (realmente esta precisión de entrenamiento no tiene mucho sentido siendo rigurosos, ya que es simplemente una idea de cómo puede ser la

precisión de validación; Tensorflow hace esta aproximación, al mismo tiempo que se recalibran los pesos al hacer el entrenamiento). La matriz resultante es la figura 25.

Es fácil interpretarla: si nos fijamos en la fila del 1, cada posición (columna) contiene la cantidad de veces que ha sido confundida por el valor de la columna correspondiente. En este preciso caso, el 1 ha sido confundido una vez por el 0, el 5, el 6 y el 8 muy pocas veces. Pero ha sido acertado 1130 veces, afortunadamente.

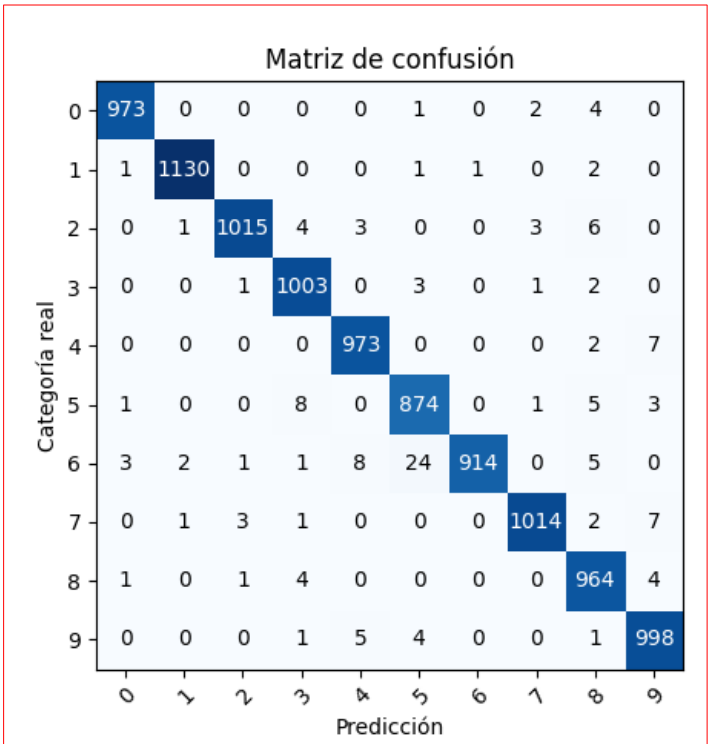
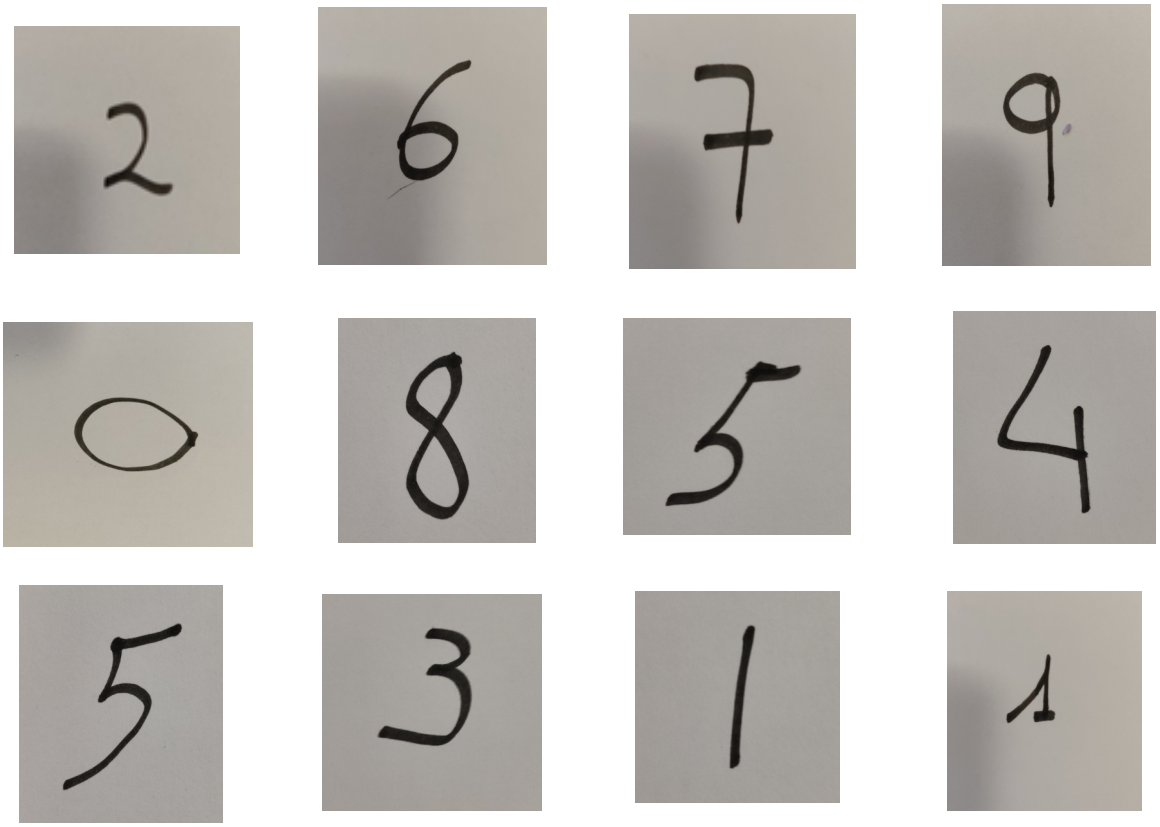


Figura 25: Matriz de confusión (creación propia)

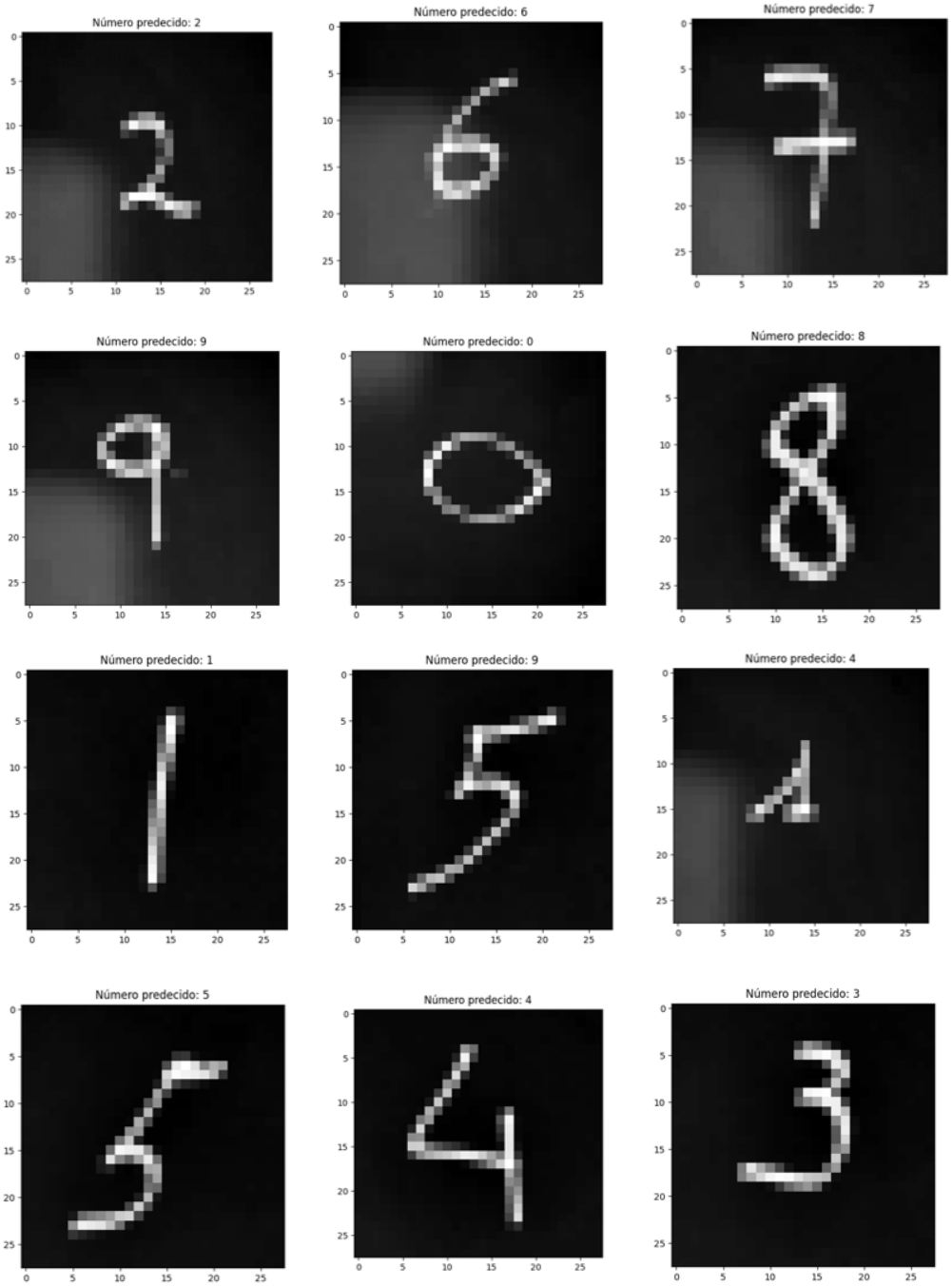
3.8. Mis números

Dado el éxito con los datos de validación, queda cumplir con uno de los objetivos del trabajo, ver cómo interpreta el modelo unos números escritos por mí, a mano. Para hacer esto, escribiré 12 números del 0 al 9,

les tomaré una foto, y guardaré estas en una carpeta. Luego, el archivo *evaluar_local.py*, ya será capaz de abrir cada una de las fotos, reducir el tamaño a 28 x 28 píxeles, pasarla a escala de grises, e interpretarla. Las fotos son las siguiente:



Al ser procesados por el modelo, resultan las siguientes imágenes con las predicciones:


Número predecido: 2
Número predecido: 6
Número predecido: 7
Número predecido: 9
Número predecido: 0
Número predecido: 8
Número predecido: 1
Número predecido: 9
Número predecido: 4
Número predecido: 5
Número predecido: 4
Número predecido: 3

Como se puede apreciar, la CNN ha detectado bien 10 de 12 dígitos, fallando 2, así teniendo una precisión del 83%. De los que ha fallado, la última imagen (el 1) ha podido ser malinterpretada seguramente porque las dos formas comunes de escribir el 1 (una con una raya y la otra más elaborada) son muy diferentes. Si este fuese un proyecto profesional, esta información se la tendríamos que aportar al modelo.

4. Conclusión

Considero haber cumplido con la desmitificación de la gran complejidad que se le otorga a procesos como los son la detección de objetos: la información ha sido suficientemente sintetizada.

- Durante el desarrollo del trabajo, he podido satisfacer mi curiosidad sobre cómo funciona una red neuronal. ¿Cuáles conceptos consideraría principales? Principalmente los del aprendizaje: clasificadores lineales, el algoritmo del perceptrón, el descenso del gradiente y la retropropagación de error (backpropagation).
- La esencia del trabajo no se limita únicamente a la detección de dígitos del 0 al 9. Entender cómo una red neuronal procesa una imagen, es fundamental para luego ver cómo se procesan los vídeos, o incluso las grabaciones a tiempo real.
- La finalidad de trabajar con redes neuronales artificiales es encontrar el tan buscado valor correcto para los pesos que hacen clasificar un objeto en una clase u otra. Para hacer esto, lo ideal es disponer de todas las muestras que estén relacionadas, pero no tengo la capacidad de procesar los millones de datos así que siempre tendremos un error mínimo. Aun así, he podido demostrar que el reconocimiento de objetos se puede hacer sin necesidad de

disponer de equipos masivamente costosos, como lo eran hace décadas, teniendo resultados considerablemente correctos.

- La potencia computacional ha sido clave para poder solventar más efectivamente un reto que solo había sido abordado por las matemáticas aplicadas.
- Paralelamente, he podido expandir mi rango de conocimientos sobre la programación de redes neuronales mediante las tecnologías de *Tensorflow* y *Keras*. Cabe destacar en este mundo, el haber aprendido a trabajar con diferentes aplicaciones, hace mucho más fácil el aprendizaje de otros lenguajes de programación o librerías complejas, por lo que haber realizado este trabajo también me ha supuesto poder moverme con mucho más dinamismo a la hora de realizar un proyecto relacionado. En este caso he trabajado con números físicos, pero esto se puede extender a objetos realmente complejos, como la identificación de caras.

4.1. Reflexión sobre el proyecto

Una vez manifestado el rendimiento que ha tenido el modelo informático, me gustaría exponer las razones por las que he llevado a cabo el proyecto así. Evidentemente quería llevar a cabo un modelo de red neuronal, para la detección de objetos en imágenes o vídeos. En un principio iba a analizar objetos mucho más complejos que unos caracteres, como pueden ser diferentes tipos de especies de plantas, las cuales tienen muchas particularidades y elementos comunes, como la existencia de pétalos, el color de estos, el largo tallo, entre otros. Como analizar objetos más complejos, requeriría de un poder computacional más grande. Como hemos visto, una cantidad mayor de variables a

estudiar está directamente relacionado al número de capas que mi red neuronal debe tener para tener un rendimiento final decente.

Por ello tenía pensado hacer estos modelos a disposición de un equipo con buenos componentes informáticos, o incluso tener un acceso al superordenador MareNostrum 4. Sin embargo, dada la situación de pandemia, no pude recurrir a otro equipo aparte del mío, con el cual tuve errores al procesar unas pruebas exigentes de una red neuronal hipotética. Por esta razón, decidí limitar la aplicación de mi proyecto, hacer un reconocimiento de objetos relativamente sencillo como es el de caracteres del 0 al 9, para centrarme exhaustivamente en el funcionamiento de este modelo (la teoría).

En cuanto al alcance del uso de redes neuronales artificiales y la mejora en el medio ambiente que puede suponer, hay mucho que hablar al respecto. Hemos de considerar que el Aprendizaje Profundo (*deep learning*) justamente se caracteriza por una intenso trabajo en cuanto a cálculos matemáticos. Así pues, el gasto energético es mayor, por lo que requeriríamos de un gran suministro eléctrico cuando hablamos de ordenadores muy potentes; son ellos los que calculan los pesos de las redes neuronales realmente complejas de nuestra realidad.

Aún así, los beneficios que proporciona una implementación considerable de las redes neuronales en muchos sectores de las industrias, por ejemplo, son extraordinarios. Por ejemplo, en las fábricas de procesamiento de materias primas, implementar un sensor que sea capaz de detectar residuos dañinos, significaría un mayor aprovechamiento de las materias primas. Por tanto, para la misma obtención de madera, por ejemplo, necesitaríamos menos recursos iniciales, menos explotación forestal; podemos extrapolar esta situación a las industrias de compuestos químicos. Por otra parte, las redes neuronales convolucionales son muy efectivas para la interpretación de

diferentes eventos en una imagen. Entonces aplicándolo a la elaboración de rutas dado un perfil topográfico (con desniveles en el relieve de una zona) o dado un sistema de carreteras, se podrían encontrar maneras eficientes de ir de un punto A a un punto B (aunque aquí ya entraríamos en el terreno de la algorítmica). Esto podría reducir la utilización de recursos al realizar un menor desplazamiento, por ejemplo en combustible.

Aunque no pretendo realizar un análisis económico de la implementación de redes neuronales, quiero dejar ver que el potencial que tienen es enorme, y se pueden aplicar a más campos de los que he mencionado, como la medicina (detección de enfermedades en el ADN) o la ciencia (desarrollo de nuevas teorías físicas).

5. Bibliografia

- (1) Rojas, R. (1996). *The* ***Backpropagation*** *Algorithm* [PDF]. Berlín: Springer-Verlag.
- (2) Buscema, M. (1998). *Back Propagation Neural Networks* [PDF]. PubMed.
- (3) Johnson, J. (2017). *Lecture 4:* ***Backpropagation*** *and Neural Networks*. Lecture presented at Assignment 1.
- (4) Nielsen, M. (2019). *Neural Networks and Deep Learning* [PDF].
- (5) V, P. (2016, December 09). ***Backpropagation*** - How Neural Networks Learn Complex Behaviors. Retrieved August/September, 2020, from **https://medium.com/autonomous-agents/*backpropagation*-how-neural-networks-learn-complex-behaviors-9572ac161670**
- (6) Zhou, V. (2019, August 8). CNNs, Part 1: An Introduction to Convolutional Neural Networks. Retrieved September 15, 2020, from **https://victorzhou.com/blog/intro-to-cnns-part-1/**
- (7) J.amatrodrigo@gmail.com, J. (18, September). Algoritmo Perceptrón. Retrieved September 02, 2020, from **https://www.cienciadedatos.net/documentos/50_algoritmo_perceptron**
- (8) Caparrini, F., & Work, W. (n.d.). Breve Historia de la Inteligencia Artificial. Retrieved August/September, 2020, from **http://www.cs.us.es/~fsancho/?e=221**
- (9) Caparrini, F., & Work, W. (n.d.). Aprendizaje Supervisado y No Supervisado. Retrieved October 27, 2020, from **http://www.cs.us.es/~fsancho/?e=77**

5.1. Referencias de las figuras

(Las figuras no referenciadas aquí son de creación propia)

- Figura 1: Clustering K-means [Digital image]. (n.d.). Consultado el 3 de mayo del 2020, desde: **https://upload.wikimedia.org/wikipedia/commons/thumb/e/e5/KMeans-Gaussian-data.svg/434px-KMeans-Gaussian-data.svg.png**
- Figura 7: Esquema neurona biológica y artificial [Digital image]. (n.d.). Consultado el 3 de mayo del 2020, desde: **http://www.cs.us.es/~fsancho/images/2019-12/artneur.gif**
- Figura 8: Multi-Layer Neural Network-Vector-Blank [Digital image]. (n.d.). Consultado el 15 de mayo del 2020, desde: **https://commons.wikimedia.org/wiki/File:Multi-Layer_Neural_Network-Vector-Blank.svg**
- Figura 9: Optimización gradiente de descenso [Digital image]. (n.d.). Consultado el 23 de junio del 2020, desde: **https://www.researchgate.net/profile/Fernando-De-La-Rosa/publication/308783857/figure/fig1/AS:530004720013312@1503374380809/Optimizacion-gradiente-de-descenso4-Este-gradiente-es-hallado-despues-de-ejecutar-el.png**
- Figura 10: Descenso de gradiente [Digital image]. (n.d.). Consultado el 23 de junio del 2020, desde:

https://ichi.pro/assets/images/max/724/1*FUL7K7JEht_AzaQCYadfGA.png

- Figura 13: Clustering K-means Iris [Digital image]. (n.d.). Consultado el 23 de junio del 2020, desde: **http://3.bp.blogspot.com/-8mFiw1nFfUo/T4NNE5LK6NI/AAAAAAAAAoA/i8Og4oFugIg/s1600/p1.png**
- Figura 14: Descomposición de una imagen en matrices [Digital image]. (n.d.). Consultado el 24 de junio del 2020, desde: **https://i0.wp.com/www.aprendemachinelearning.com/wp-content/uploads/2018/11/cnn-02.png?w=854&ssl=1**
- Figura 15 y Figura 16 (recortes de una misma imagen): Filtro Sobel Vertical [Digital image]. (n.d.). Consultado el 24 de junio del 2020, desde: **https://victorzhou.com/static/44a1ff59f9a2c7f62cf9f56a8398efd0/fa73e/lenna%2Bvertical.webp**
- Figura 17 y Figura 18 (recortes de una misma imagen): Filtro Sobel Horizontal [Digital image]. (n.d.). Consultado el X, desde: **https://victorzhou.com/static/342e8364a392ac2cbcd4ecc7b9aacaa1/fa73e/lenna%2Bhorizontal.webp**
- Figura 19: Subsampling y maxpooling [Digital image]. (n.d.). Consultado el 12 de agosto del 2020, desde: **https://i1.wp.com/www.aprendemachinelearning.com/wp-content/uploads/2018/11/cnn-05.png?w=395&ssl=1**
- Figura 20: Primera convolución [Digital image]. (n.d.). Consultado el 12 de agosto del 2020, desde:

https://i2.wp.com/www.aprendemachinelearning.com/wp-content/uploads/2018/11/cnn-06.png?w=960&ssl=1

- Figura 21: Segunda convolución [Digital image]. (n.d.). Consultado el 12 de agosto del 2020, desde: **https://i0.wp.com/www.aprendemachinelearning.com/wp-content/uploads/2018/11/cnn-07.png?w=960&ssl=1**
- Figura 22: Arquitectura Red Neuronal Convolucional [Digital image]. (n.d.). Consultado el de agosto del 2020, desde: **https://i0.wp.com/www.aprendemachinelearning.com/wp-content/uploads/2018/11/CNN-08.png?w=943&ssl=1**

6. Anexos

- Repositorio online del proyecto:

  https://github.com/gmorams/CNN-Reconocimiento-Objetos

6.1. Código de la creación de la red neuronal:

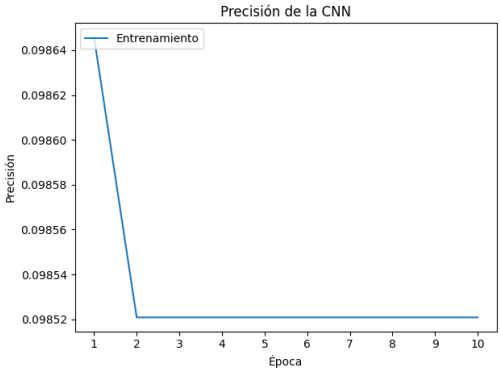
```
from __future__ import division
import tensorflow.keras as keras
import matplotlib.pyplot as plt
import numpy as np
import copy
import random
from tensorflow.keras.utils import to_categorical  #para pasara a one-hot
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation, Conv2D,
MaxPooling2D, Flatten
from tensorflow.keras.optimizers import SGD

##Importamos el conjunto de datos de Keras
from tensorflow.keras.datasets import mnist
##Cargamos los datos de entrenamiento y etiquetas correspondientes, y
los datos de validacion y etiquetas correspondientes
(x_train, y_train), (null, null) = mnist.load_data(path='data')

#Transformamos las imagenes a vector, añadiendo una dimension (antes
estaban en 6000,28,28)
x_train = x_train.reshape(60000,28,28,1)
#x_test = x_test.reshape(10000,28,28,1)

x_train = x_train / 255
```

```
#x_test = x_test / 255

#pasar a formato one_hot (de 3 a [0,0,0,1])
y_train = to_categorical(y_train, 10)
#y_test = to_categorical(y_test, 10)

###Creacion de la CNN
modelo = Sequential()
# CONV1 Y MAX-POOLING1
modelo.add(Conv2D(filters=6, kernel_size=(5,5), activation='relu',
input_shape=(28,28,1)))
modelo.add(MaxPooling2D(pool_size=(2,2)))
# CONV2 Y MAX-POOLING2
modelo.add(Conv2D(filters=16, kernel_size=(5,5), activation='relu'))
modelo.add(MaxPooling2D(pool_size=(2,2)))
# Aplanar, FC1, FC2 y salida
modelo.add(Flatten())
modelo.add(Dense(120,activation='relu'))
modelo.add(Dense(84,activation='relu'))
modelo.add(Dense(10,activation='softmax'))

#COMPILACION
modelo.compile(loss='categorical_crossentropy', optimizer='adam',
metrics=['accuracy'])
#TRAIN
nepochs = 25
history = modelo.fit(x_train,y_train,batch_size=128,epochs=nepochs,
validation_split=0.2)
#modelo.fit(x_test,y_test,batch_size=128,epochs=nepochs,
validation_split=0.2)
```

```
##### ESTUDIO
acc_plot = plt.figure(1)
plt.plot(history.history['accuracy'])
plt.title('Precisión de la CNN')
plt.ylabel('Precisión')
plt.xlabel('Época')
plt.xticks([x for x in range(nepochs)], [x + 1 for x in range(nepochs)])
plt.legend(['Entrenamiento'], loc='upper left')

loss_plot = plt.figure(2)
plt.plot(history.history['loss'])
plt.title('Error en la CNN')
plt.ylabel('Error')
plt.xlabel('Época')
plt.xticks([x for x in range(nepochs)], [x + 1 for x in range(nepochs)])
plt.legend(['Entrenamiento'], loc='upper left')
acc_plot.show()
#loss_plot.show()

#SCORE
score = modelo.evaluate(x_train, y_train, verbose=0)
#print('test loss:', score[0])
print('test accuracy:', score[1]*100)

###GUARDAR MODELO
# serializar el modelo a JSON
modelo_json = modelo.to_json()
with open("model.json", "w") as json_file:
    json_file.write(modelo_json)
```

```
# serializar los pesos a HDF5
modelo.save_weights("model.h5")
print("Modelo Guardado!")
```

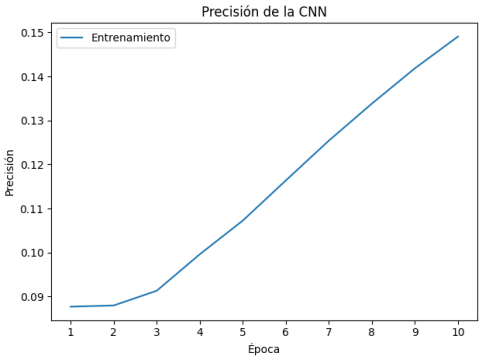
6.2. Pruebas modelo CNN



Prueba 0:

- Función de activación: Exponencial
- Función de error: Error Cuadrático Medio
- Dimensiones *kernel*: 5x5
- Número de Épocas: 10
- Optimizador: SGD

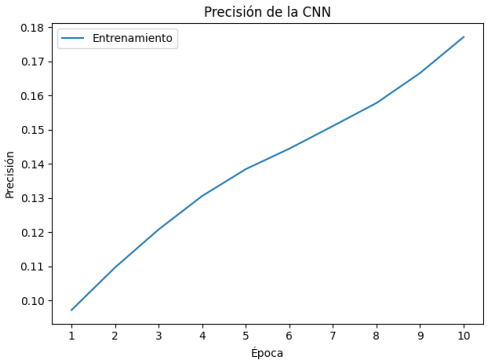
- Precisión final: 9.871%



Prueba 1:

- Función de activación: ReLU
- Función de error: Error Cuadrático Medio
- Dimensiones *kernel*: 5x5
- Número de Épocas: 10
- Optimizador: SGD

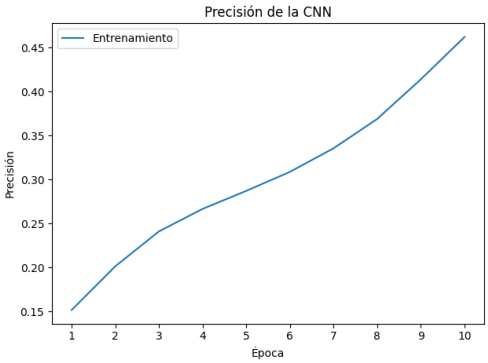
- Precisión final: 15,408%



Prueba 2:

- Función de activación: ReLU
- Función de error: Error Cuadrático Medio
- Dimensiones *kernel*: 3x3
- Número de Épocas: 10
- Optimizador: SGD

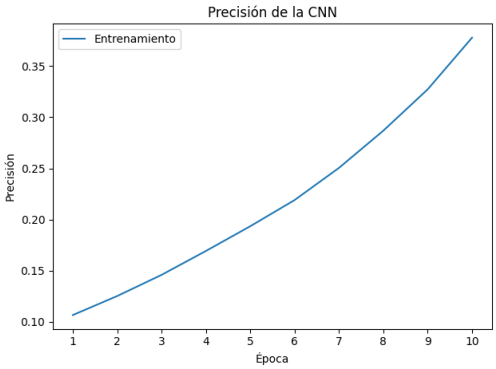
- Precisión final: 18,355%



Prueba 3:

- Función de activación: Tanh
- Función de error: Error Cuadrático Medio
- Dimensiones *kernel*: 5x5
- Número de Épocas: 10
- Optimizador: SGD

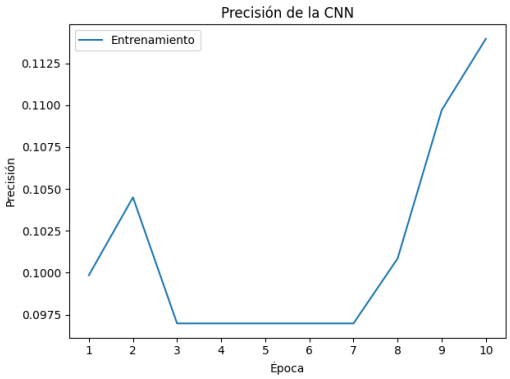
- Precisión final: 49,465%



Prueba 4:

- Función de activación: Tanh
- Función de error: Error Cuadrático Medio
- Dimensiones *kernel*: 3x3
- Número de Épocas: 10
- Optimizador: SGD

- Precisión final: 40,248%

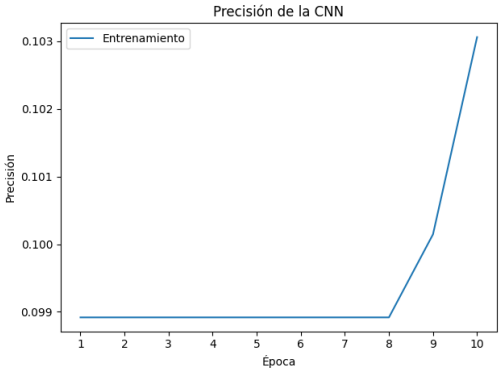


Prueba 5:

- Función de activación: Sigmoide
- Función de error: Error Cuadrático Medio
- Dimensiones *kernel*: 5x5
- Número de Épocas: 10
- Optimizador: SGD

- Precisión final: 11,237%

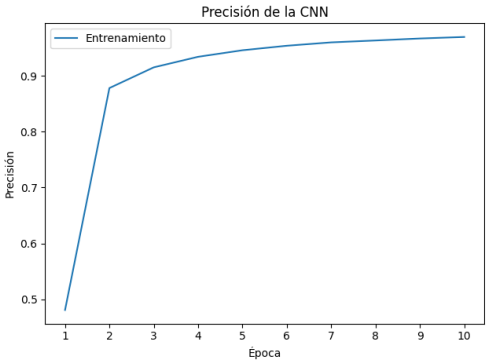
-



Prueba 6:

- Función de activación: Sigmoide
- Función de error: Error Cuadrático Medio
- Dimensiones *kernel*: 3x3
- Número de Épocas: 10
- Optimizador: SGD

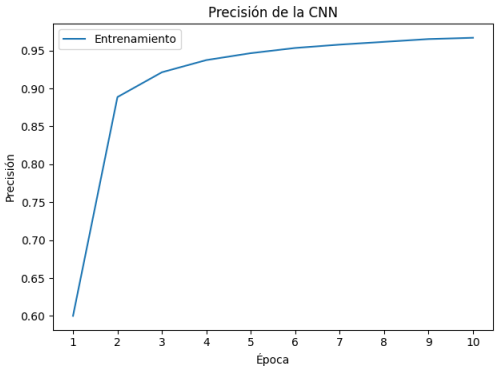
- Precisión final: 10,441%



Prueba 7:

- Función de activación: ReLU
- Función de error: Entropía Cruzada Categórica
- Dimensiones *kernel*: 5x5
- Número de Épocas: 10
- Optimizador: SGD

- Precisión final: 96,333%



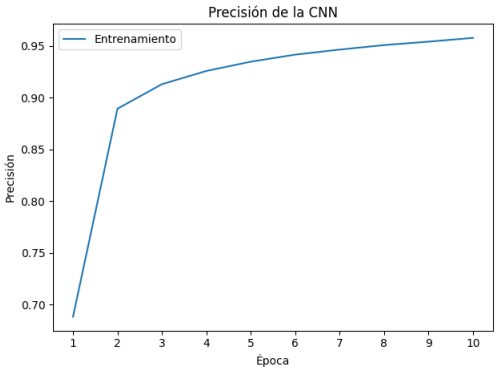
Prueba 8:

- Función de activación: ReLU
- Función de error: Entropía Cruzada Categórica
- Dimensiones *kernel*: 3x3
- Número de Épocas: 10
- Optimizador: SGD

-

Precisión final: 96,67%

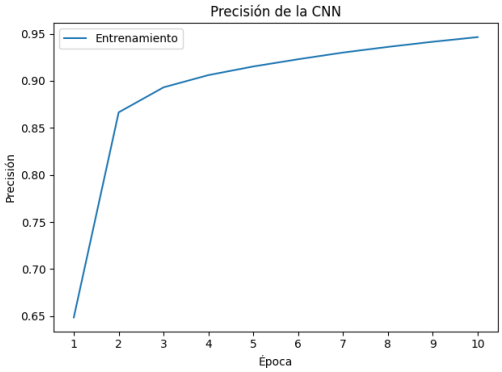
Prueba 9:



- Función de activación: Tanh
- Función de error: Entropía Cruzada Categórica
- Dimensiones *kernel*: 5x5
- Número de Épocas: 10
- Optimizador: SGD

- Precisión final: 95,96%

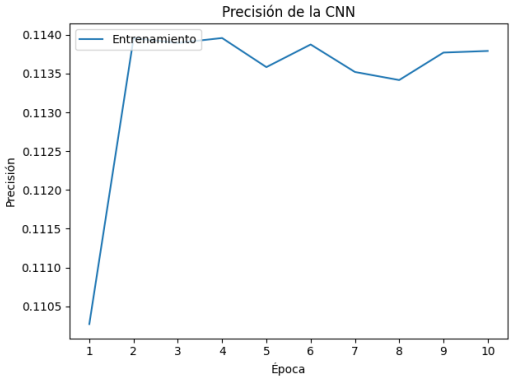
Prueba 10:



- Función de activación: Tanh
- Función de error: Entropía Cruzada Categórica
- Dimensiones *kernel*: 3x3
- Número de Épocas: 10
- Optimizador: SGD

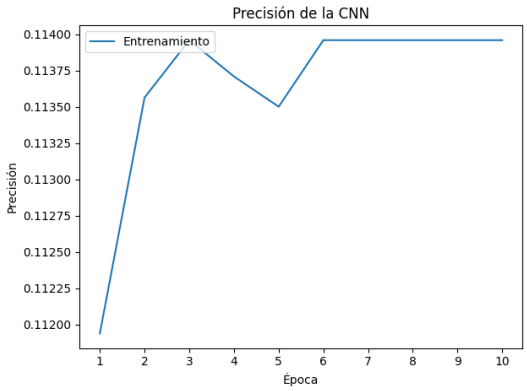
- Precisión final: 94,98%

Prueba 11:



- Función de activación: Sigmoide
- Función de error: Entropía Cruzada Categórica
- Dimensiones *kernel*: 5x5
- Número de Épocas: 10
- Optimizador: SGD

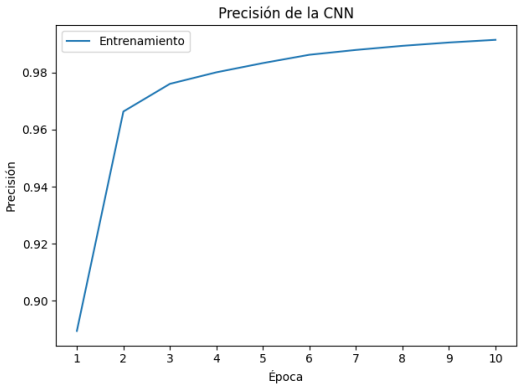
- Precisión final: 11,23%



Prueba 12:

- Función de activación: Sigmoide
- Función de error: Entropía Cruzada Categórica
- Dimensiones *kernel*: 3x3
- Número de Épocas: 10
- Optimizador: SGD

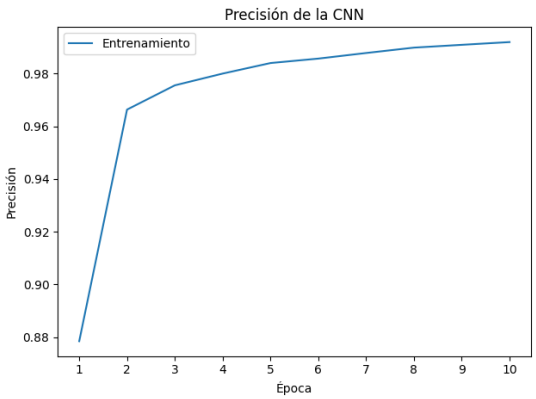
- Precisión final: 11,24%



Prueba 13:

- Función de activación: ReLU
- Función de error: Error Cuadrático Medio
- Dimensiones *kernel*: 5x5
- Número de Épocas: 10
- Optimizador: Adam

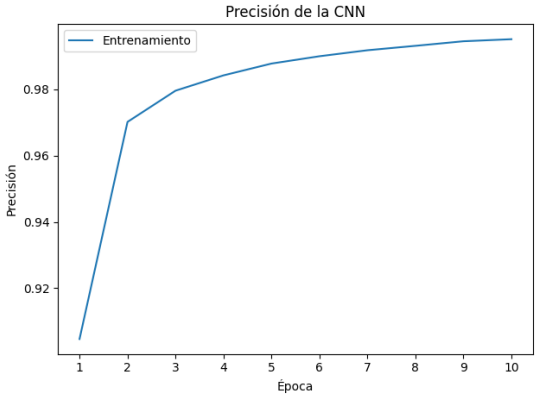
- Precisión final: 99,31%



Prueba 14:

- Función de activación: ReLU
- Función de error: Error Cuadrático Medio
- Dimensiones *kernel*: 3x3
- Número de Épocas: 10
- Optimizador: Adam

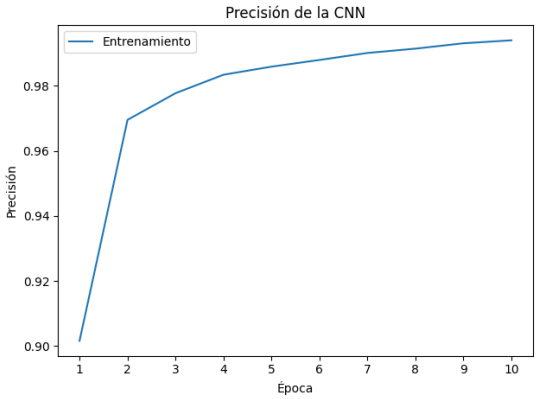
- Precisión final: 99,27%



Prueba 15:

- Función de activación: Tanh
- Función de error: Error Cuadrático Medio
- Dimensiones *kernel*: 5x5
- Número de Épocas: 10
- Optimizador: Adam

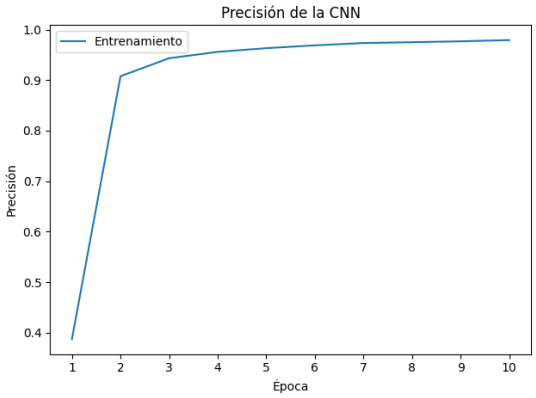
- Precisión final: 99,37%



Prueba 16:

- Función de activación: Tanh
- Función de error: Error Cuadrático Medio
- Dimensiones *kernel*: 3x3
- Número de Épocas: 10
- Optimizador: Adam

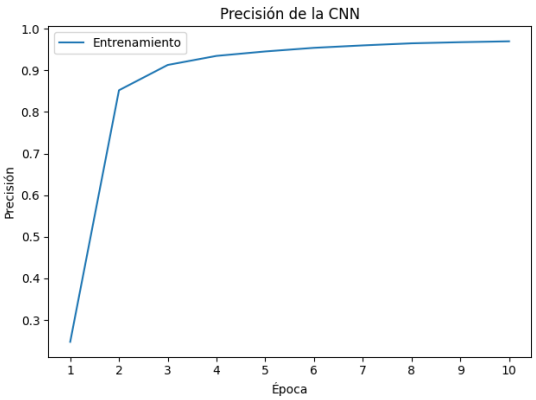
- Precisión final: 99,11%



Prueba 17:

- Función de activación: Sigmoide
- Función de error: Error Cuadrático Medio
- Dimensiones *kernel*: 5x5
- Número de Épocas: 10
- Optimizador: Adam

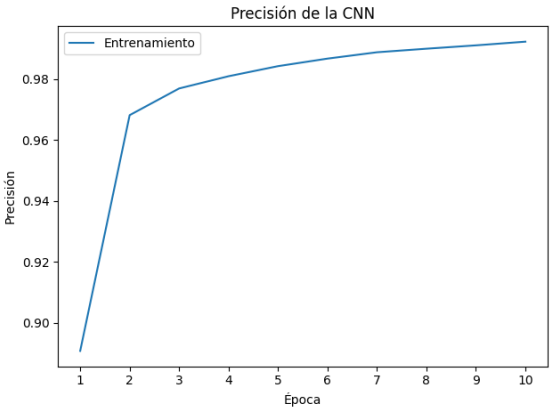
- Precisión final: 97,85%



Prueba 18:

- Función de activación: Sigmoide
- Función de error: Error Cuadrático Medio
- Dimensiones *kernel*: 3x3
- Número de Épocas: 10
- Optimizador: Adam

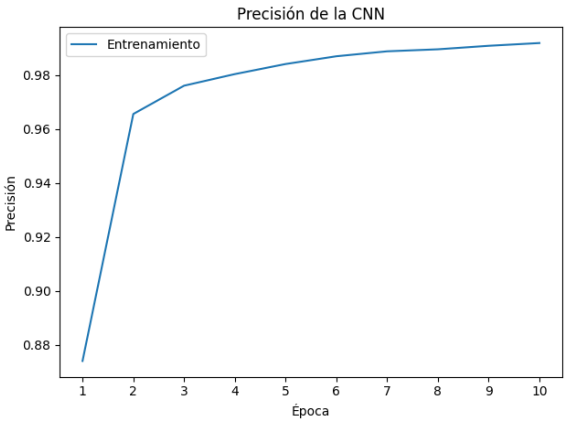
- Precisión final: 97,24%



Prueba 19:

- Función de activación: ReLU
- Función de error: Entropía Cruzada Categórica
- Dimensiones *kernel*: 5x5
- Número de Épocas: 10
- Optimizador: Adam

- Precisión final: 99,40%

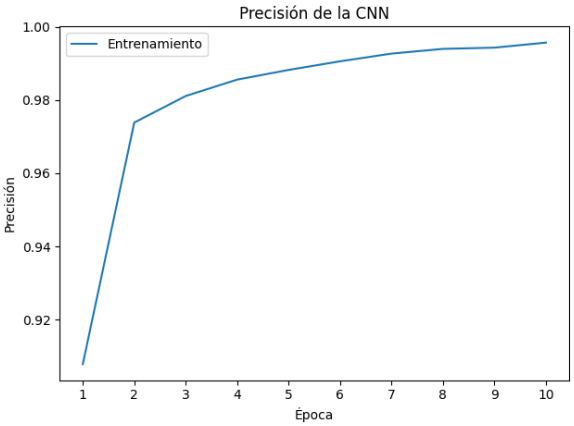


Prueba 20:

- Función de activación: ReLU
- Función de error: Entropía Cruzada Categórica
- Dimensiones *kernel*: 3x3
- Número de Épocas: 10

- Optimizador: Adam

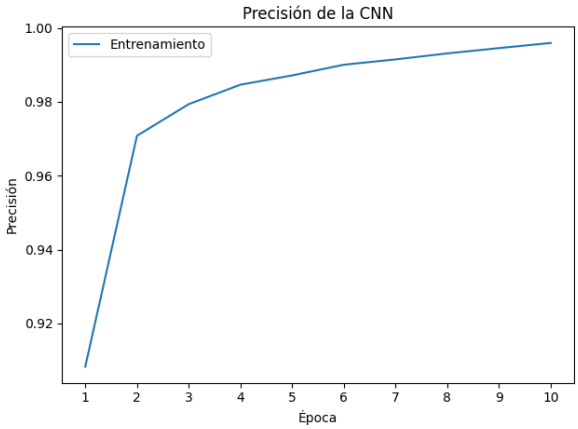
- Precisión final: 99,24%



Prueba 21:

- Función de activación: Tanh
- Función de error: Entropía Cruzada Categórica
- Dimensiones *kernel*: 5x5
- Número de Épocas: 10
- Optimizador: Adam
-

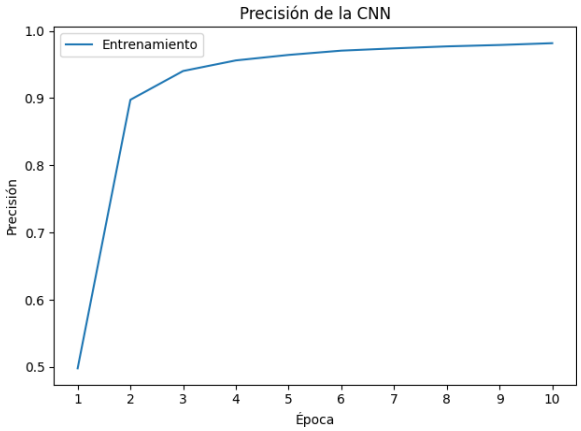
- Precisión final: 99,14%



Prueba 22:

- Función de activación: Tanh
- Función de error: Entropía Cruzada Categórica
- Dimensiones *kernel*: 3x3
- Número de Épocas: 10
- Optimizador: Adam
-

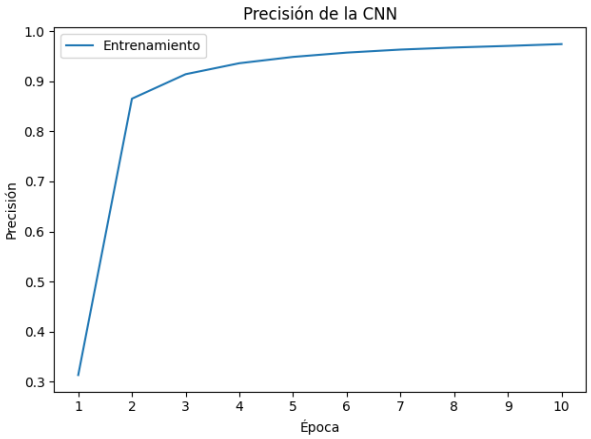
- Precisión final: 99,02%



Prueba 23:

- Función de activación: Sigmoide
- Función de error: Entropía Cruzada Categórica
- Dimensiones *kernel*: 5x5
- Número de Épocas: 10

- Optimizador: Adam
- Precisión final: 98,31%



Prueba 24:

- Función de activación: Sigmoide
- Función de error: Entropía Cruzada Categórica
- Dimensiones *kernel*: 3x3
- Número de Épocas: 10
- Optimizador: Adam

- Precisión final: 97,54%